Algorithms for the analysis of molecular sequences

Vayani, Fatima

Awarding institution:
King's College London

Algorithms for the Analysis of Molecular Sequences

Fatima Vayani

Supervisor: Prof. C. S. Iliopoulos

Dr. S. P. Pissis

Department of Informatics

King's College London

This dissertation is submitted for the degree of

Doctor of Philosophy in Computer Science

September 2018

I dedicate this thesis to my parents, Nasim and

Abdul Ghaffar, and my neices: Aaishah, Ameera, and Mariam.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified. This dissertation contains fewer than 100,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Fatima Vayani

September 2018

Acknowledgements

In the name of God, the Most Gracious and Most Merciful.

I would like to begin by thanking my principal supervisor, **Prof. Costas S. Iliopoulos**, for training me to become a well-rounded researcher; for trusting me; and for providing me with innumerable invaluable research opportunities. My experience as a student would simply not have been as rich without him, and his generosity will not be forgotten.

I also thank my second supervisor, **Dr. Solon P. Pissis**, for helping me to understand theoretical computer science; for his time and patience as a teacher; and for his inspiration as a passionate researcher. His breadth and depth of knowledge, and ability to articulate complex concepts so eloquently, are qualities I endeavour to attain.

Dr. Manal Mohamed has been a close collaborator throughout my time as a student. I thank her warmly for dedicating her time to the countless hours of our passionate discussions; and for her advice, feedback, and help on the various projects we have collaborated on.

I thank the Engineering and Physical Sciences Research Council for awarding me with a three-year studentship, and all of the co-authors with whom I have collaborated over the years. I also thank the Department of Informatics and the Faculty of Natural and Mathematical Sciences at King's College London for providing me with an enjoyable and comfortable working environment. In particular, I thank **Lucy Ward**, **Dr. Elizabeth Black**, **Prof. Maxime Crochemore**, and **Dr. Sanjay Modgil**.

I would like to thank my colleagues and friends for their support throughout my time as a doctoral candidate, and for all of the great memories. In particular, I thank **Abbie**, **Ahmad**, **Ana**, **Annie**, **Amal**, **Atif**, **Burayha**, **Chang**, **Hayam**, **Hessa**, **Idelès**, **Jia**, **Joynob**, **Lorraine**, **Mai**, **Memoona**, **Misha**, **Miznah**, **Mudhi**, **Panos**, **Ritu**, **Sara**, **Senija**, **Shokryah**, **Shuchi**, **Sisi**, **Steven**, **Sweta** and **Wole**.

Finally, and most importantly, I would like to thank my parents for their exceptional level of patience; for being my pillars of support; and for knowing I am grateful even when I do not show it.

I have grown immensely as a person and, I believe, have successfully completed my mutation from biologist to computer scientist. I hope that this thesis is a testament to that.

Abstract

DNA sequencing is the translation of molecular structure into a human- and machine-readable format: a sequence, or *string*, of letters. The exponential growth of data (from DNA, RNA, and proteins) produced by biotechnology has resulted in two major scientific questions. First, what conclusions can we draw from *all* of the data that we have? Second, how can we do this in an *efficient* manner? The answers to these questions are where computer science and biological science meet: in the research field of *bioinformatics*. The obvious beauty of the aforementioned fields of study is their resemblance to *stringology*, the analysis of strings.

The research presented in this thesis lies within the intersection of computational molecular biology and stringology. Specifically, the aim was to design string-processing algorithms to analyse molecular sequences, in order to aid and enhance biological research. This thesis is an exploration of three important concepts in molecular biology: circular molecules, sequence motifs, and pan-genomes. In Chapter 2, we study the problem of accuracy when aligning two linear sequences obtained from circular molecular structures. Chapter 3 focuses on common, and thus biologically important, patterns found in molecular sequences. Lastly, in Chapter 4, we consider the complexities of handling pan-genomic data.

Publications

Cited in thesis:

[15] Barton, C., Iliopoulos, C.S., Kundu, R., Pissis, S.P., Retha, A. and **Vayani, F.**, 2015, June. Accurate and efficient methods to improve multiple circular sequence alignment. In *International Symposium on Experimental Algorithms* (pp. 247-258). Springer, Cham.

[67] Grossi, R., Iliopoulos, C.S., Liu, C., Pisanti, N., Pissis, S.P., Retha, A., Rosone, G., **Vayani, F.** and Versari, L., 2017. On-line pattern matching on similar texts. In *LIPIcs-Leibniz International Proceedings in Informatics* (Vol. 78). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[68] Grossi, R., Iliopoulos, C.S., Mercaş, R., Pisanti, N., Pissis, S.P., Retha, A. and **Vayani, F.**, 2015, September. Circular sequence comparison with q-grams. In *International Workshop on Algorithms in Bioinformatics* (pp. 203-216). Springer, Berlin, Heidelberg.

[69] Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A. and **Vayani, F.**, 2016. Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11(1), p.12.

[85] Iliopoulos, C.S., Mohamed, M., Pissis, S.P. and **Vayani, F.**, 2018, October. Maximal motif discovery in a sliding window. In *International Symposium on String Processing and Information Retrieval* (pp. 191-205). Springer, Cham.

Not cited in thesis:

Crochemore, M., Iliopoulos, C.S., Kundu, R., Mohamed, M. and **Vayani, F.**, 2016. Linear algorithm for conservative degenerate pattern matching. *Engineering Applications of Artificial Intelligence*, 51, pp.109-114.

Brankovic, L., Iliopoulos, C.S., Kundu, R., Mohamed, M., Pissis, S.P. and **Vayani, F.**, 2016. Linear-time superbubble identification algorithm for genome assembly. *Theoretical Computer Science*, 609, pp.374-383.

Iliopoulos, C.S., Kundu, R., Mohamed, M. and **Vayani, F.**, 2016, March. Popping superbubbles and discovering clumps: recent developments in biological sequence analysis. In *International Workshop on Algorithms and Computation* (pp. 3-14). Springer, Cham.

*N*ote that alphabetical ordering of authorship is the norm in this area of research.

Table of contents

Chapter 1

Introduction

1.1 Biological Motivation

Since the award of the Nobel Prize for the elucidation of the structure of deoxyribonucleic acid (*DNA*) in 1962 [176], advancements in the field of genetics have transformed our understanding of medicine, physiology and botany, to name a few. This is because DNA is the code which ultimately defines the structure and function of every molecule inside every living (humans, animals, plants, and bacteria) and non-living organism (viruses). The most significant advancement was the establishment of highly efficient techniques within biotechnology which enable us to study the structure of DNA molecules at great levels of detail and accuracy. Many of these modern technologies, known as Next-Generation DNA Sequencers (*NGS*), are based on the very first technique developed by Sanger *et al.* in 1977 [149]. DNA sequencing is the translation of molecular structure into a human- and machine-readable format: a sequence, or *string*, of letters. The output of NGS technologies varies greatly in quality and, therefore, usefulness. The former is constantly being improved through the introduction of new technologies, but even more so through the design of sophisticated algorithms to rectify errors in the output. As advanced technologies are being

developed, the cost of sequencing DNA is decreasing, and the volume of data is increasing. The large-scale study of DNA sequences is known as *genomics*.

Similarly, *proteomics* is the term used to describe the large-scale study of proteins. The structure and function of proteins result from the DNA that encode them. The techniques by which proteins are sequenced are very different from DNA, but result in similar output: a string of letters. The two most important differences between DNA and protein sequences are their alphabets and what each letter in the alphabet represents. In DNA, the alphabet comprises `A`, `C`, `G`, and `T`, where each represents a monomer called a *nucleotide*. In proteins, the size of the alphabet is twenty, where each letter represents a monomer called an *amino acid*.

The final molecular piece of the puzzle that is the central dogma of molecular biology is ribonucleic acid (*RNA*). In many cellular mechanisms, RNA is simply a messenger molecule existing between DNA and proteins. However, there are several circumstances in which RNA has specific crucial functions.

The exponential growth of data (from DNA, RNA, and proteins) produced by biotechnology has resulted in two major scientific questions. First, what conclusions can we draw from *all* of the data that we have? Second, how can we do this in an *efficient* manner? The answers to these questions are where computer science and biological science meet: in the research field of *bioinformatics*.

The obvious beauty of the aforementioned fields of study is their resemblance to *stringology*. That is, the process of analysing a sentence from a book and a DNA sequence is exactly the same. The difference, however, lies in the desired outcome of the analysis.

Due to this resemblance, the research described in this thesis is mutually beneficial to both fields of research, as algorithms designed to analyse molecular sequences could equally be given other strings as input. Stringology, also known as *combinatorics on words*, involves the design of algorithms to analyse strings (mathematics) and their implementation (computer science).

The research presented in this thesis lies within the intersection of computational molecular biology and stringology. Specifically, the aim was to design string-processing algorithms to analyse molecular sequences, in order to aid and enhance biological research. To that end, many of these algorithms have been or will be implemented as open-source tools which can be used by biologists or bioinformaticians.

1.2 Thesis Overview

This thesis is an exploration of three important concepts in molecular biology: circular molecules, sequence motifs, and pan-genomes. Due to the faster pace of algorithm design and implementation, compared to biological research conducted in a laboratory, the main body of this thesis is a collection of several chapters based on published, or accepted, papers. These are preceded by a section of definitions which are required to gain a thorough understanding of the work from an inter-disciplinary perspective (Section 2.2). Specifically:

- In Chapter 2, we study the problem of accuracy when aligning two linear sequences obtained from circular molecular structures.
- Chapter 3 focuses on common, and thus biologically important, patterns found in molecular sequences.
- Lastly, in Chapter 4, we consider the complexities of handling pan-genomic data.

The thesis is concluded in Chapter 5.

1.3 Definitions

1.3.1 Biological Concepts and Definitions

Following the introduction of molecular biology, this section provides some fundamental definitions and concepts.

DNA is a polymer of *nucleotides*. Nucleotides are organic molecules that share a common structure, with only one of the functional groups differing in each type of nucleotide. There exist four different nucleotides, therefore, with four different nucleobases: *Adenine*, *Guanine*, *Cytosine*, and *Thymine*. These nucleobases can be represented as single letters: `A`, `G`, `C`, and `T`. The other two functional groups in a nucleotide molecule are the phosphate and sugar groups. These are what allow nucleotides to polymerise, resulting in a DNA molecule with a repetitive structure. Specifically, the phosphate group of one nucleotide bonds to the sugar group of another nucleotide, creating the sugar-phosphate backbone of a DNA molecule. Nucleobases, or simply bases, can be classified in to two groups, due to their molecular structures: purines (`A` and `G`) and pyrimidines (`T` and `C`).

Due to the delicate structure of bases, and the importance of the information their sequences encode, the structure of DNA must be extremely stable, yet flexible enough to be replicated. This balance is achieved by *base pairing*. Each pair of bases in a pair of DNA polymers bond with each other to create a structure similar to a ladder, where the stiles of the ladder are the sugar-phosphate backbones, and the steps are the pairs of bases bonded to each other. Importantly, `A` bonds with `T`, and `G` bonds with `C`.

The third level of the structure of DNA, after the sequence of the bases and base pairing, is the helical structure of the DNA molecule when in its most stable state. The structure is known as an α-*helix* and is only unwound during very specific cellular processes. For example, the bonds between bases are broken during DNA replication and only one strand of DNA is replicated. Intuitively, only one strand is required, as the strand it pairs with is essentially the same, but with complementary bases.

Example 1. *Given a DNA sequence of* `AGCTAGCT`*, one can immediately deduce that the sequence of its complementary strand will be* `TCGATCGA`*.*

Therefore, the information stored by a DNA molecule is in the sequence of its bases.

Many sequences within DNA molecules are not yet well understood. An exception to this are *genes*. Genes are regions of DNA sequences which may ultimately encode proteins. Their structures are complex as they require specific *signals* in order to indicate that they are genes, and instruct which sequences within them should be expressed and which should be retained. Signals include start and stop signals which indicate the beginning and end of a gene. Between these signals are *introns* and *exons*, which should be retained and expressed, respectively. The structure of genes varies greatly in different organisms. Bacteria, for example, do not have intron/exon structure, and therefore the entirety of the gene is always expressed. Furthermore, the regions in between genes are yet to be completely understood. Not surprisingly, these regions are much shorter in bacteria than in humans.

The collection of genes and intergenic regions in an organism are known as its *genome*. The entire genome of an organism is stored in the nucleus of each cell in the organism. The regions of the genome that are expressed are what cause different cell types to differ. For example, a skin cell and a liver cell both contain copies of the genome, but the reason for their greatly differing functions is due to the difference in the collective expression of specific

genes. The expression of genes into proteins is achieved with the use of *ribonucleic acid (RNA)*.

RNA is extremely similar in structure to DNA, with only two major differences. First, the alphabet differs by one base, Uracil (`U`), which replaces `T` in the DNA alphabet. Therefore, the RNA alphabet is `A`, `G`, `C`, and `U`; where `A` now bonds with `U`. Second, the structure of RNA must be more flexible due to its functions. Where DNA is a highly stable and guarded storage for information, RNA molecules are less stable, more actively functional molecules. RNA molecules are single-stranded and do not form the same ladder structure as with DNA. Instead, bases in a strand of RNA form bonds with other bases in the same molecule, creating molecules of varying shapes and functions.

There are many different types of RNA and here we will divide them in to two classes: *messenger RNA (mRNA)* and non-messenger RNA. *Transcription* is the cellular mechanism by which a gene is transcribed, by an enzyme, into RNA. Specifically, mRNA is later translated into a protein structure, by a complex molecular machine, called a *ribosome*, comprised of proteins and non-messenger RNA molecules. Other types of non-messenger RNA molecules are involved in the regulation of gene expression as well as the process of protein synthesis itself. *MicroRNAs*, for example, target and label mRNAs for degradation, thereby halting the process of protein synthesis. *Transfer RNAs (tRNAs)* are molecules which assist protein synthesis.

Note that DNA and RNA are collectively known as *nucleic acids*.

Protein synthesis is a complex process, beginning with the translation of mRNA into a polymer known as a *peptide*. The mRNA molecule is read by a ribosome, and tRNAs are able to decipher which sequence of three nucleotides in the mRNA (known as *codons*) result in which amino acid. Processing mRNA one codon at a time results in the synthesis of a

polypeptide, because as each codon is deciphered by each tRNA, an enzyme causes a peptide bond to be formed between each subsequent amino acid, thereby producing a polymer of amino acids: a *polypeptide*. This sequence of amino acids is known as the *primary structure* of the protein.

There are twenty amino acids in nature, which are similar to nucleic acids in that parts of them have the same structure, and only one functional group differs, resulting in variation between them. Each amino acid can be represented by a single letter. Therefore, the protein alphabet has a size of twenty. The primary structure of a protein is extremely important to its functionality. Some specific short sequences within protein primary structures can interact with other molecules and are known as *motifs*. There are three higher levels of structure for proteins, the descriptions of which are beyond the scope of this thesis.

1.3.2 Algorithmic Tools, Properties, and Definitions

This section, generally following [43, 44, 74], provides basic definitions required to understand and design algorithms for strings.

Fundamental Definitions

A fixed *alphabet* Σ is a non-empty finite set of *letters* or *symbols* of size $|\Sigma|$. In the current context of molecular biology, we assume that the alphabet is fixed, that is, $|\Sigma| = \mathcal{O}(1)$. We define the DNA alphabet as $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. A *(deterministic) string* built on Σ is a finite sequence of symbols drawn from Σ. The *length* of a string X is denoted by $|X|$. We can think of a string X as an array $X[0..|X|-1]$. The *empty string* of length 0 is denoted by ε. The set of all strings over an alphabet Σ (including ε) is denoted by Σ^*. For two positions i and j in X, we denote by $X[i..j] = X[i]..X[j]$ the *factor* or *substring* of X that starts at position i and ends at position j; it is empty if $j < i$. For any string $X = UYV$, if $U = \varepsilon$ then Y is a

prefix of X. Similarly, if $V = \varepsilon$ then Y is a *suffix* of X. We say that Y is a *proper factor* (resp. prefix/suffix) of X if Y is a factor (resp. prefix/suffix) of X distinct from X.

We say that there is an *occurrence* of Y in X, or, simply, that Y *occurs in* X, when Y is a factor of X. We denote the set of occurrences of Y in X by $\mathsf{occ}_X(Y)$, or simply $\mathsf{occ}(Y)$ when the context is clear. We therefore denote the number of occurrences of Y in X by $|\mathsf{occ}_X(Y)|$.

Suffix, Inverse Suffix, and Longest Common Prefix Arrays

We denote by SA the *suffix array* of a string X of length $n > 0$, that is, an integer array of size n storing the starting positions of all lexicographically sorted suffixes of X. That is, for all $r \in [1,n)$, we have $X[\mathsf{SA}[r-1]\,..\,n-1] < X[\mathsf{SA}[r]\,..\,n-1]$ [108].

Let $\mathsf{lcp}(r,s)$ denote the length of the longest common prefix between $X[\mathsf{SA}[r]\,..\,n-1]$ and $X[\mathsf{SA}[s]\,..\,n-1]$, for all positions r, s on X; and 0 if they do not have a common prefix. We denote by LCP the *longest common prefix* array of X defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1,r)$, for all $r \in [1,n)$, and $\mathsf{LCP}[0] = 0$. The inverse of SA, denoted by iSA, is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $r \in [0,n)$.

Fact 1 ([59])**.** *Arrays SA, iSA, and LCP of a string X of length n can be computed in time and space $\mathcal{O}(n)$.*

Suffix Tree

Given a string X, of length $n > 0$, the *suffix tree* ST_X of X is a compact trie representing all suffixes of X. The nodes of the trie which become nodes of the suffix tree are *explicit* nodes; all other nodes are *implicit*. That is, a node v in the trie with only one outgoing edge

is implicit in the suffix tree. The root node r represents the empty string ε. Each edge of the suffix tree can be viewed as a path of implicit nodes from one explicit node to another.

Fact 2 ([54, 74, 110, 170, 179])**.** *Given a string X of length n, ST_X can be constructed in $\mathcal{O}(n)$ time and space.*

Each implicit node can be represented by a tuple $\langle \gamma, \mu, \lambda \rangle$, where μ is the number of implicit nodes skipped on the edge from explicit node γ, and the edge label ends with $\alpha = X[\lambda]$. Note that in practice, edge labels are not stored as strings; rather, they are stored as intervals of the input string. The *path-label* $P(V)$ of a node V is the concatenation of the edge labels along the path from the root to V. Henceforth, we will use $P(V)$ and V interchangeably to refer to the factor of X that V represents. The *string-depth* $D(V) = |P(V)|$ of a node V is the total number of implicit and explicit nodes in the path from the root to V.

Node V is known as a *terminal* node if its path-label is a suffix of X; that is, $P(V) = X[i..n-1]$ for some $i \in [0,n)$. Note that, each leaf node in ST_X is a terminal node. If a special letter $\$ \notin \Sigma$ is appended to X then each terminal node is necessarily a leaf node in ST_X and the suffix tree is *explicit*; otherwise, it is an *implicit suffix tree*. See Figures 1.1 and 1.2 below for simple examples of explicit and implicit suffix trees, respectively. See Figures 3.1 and 3.3 in Chapter 3 for more complex examples.

Fact 3 ([74])**.** *Using ST_X, $|\mathsf{occ}_X(V)|$ can be computed by traversing the subtree rooted at node V and counting its number of terminal nodes in $\mathcal{O}(|\mathsf{occ}_X(V)|)$ time.*

The *suffix link* from a *suffix origin*, node V, with path-label $P(V) = \alpha W$, is a pointer to a *suffix target*, node $s(V)$, path-labelled $P(s(V)) = W$, where $\alpha \in \Sigma$ and W is a factor of X. See Figure 3.1 for examples.

Figure 1.1 The explicit suffix tree of the string $X = \texttt{CTATTAGG}$ concatenated with $\$ \notin \Sigma$. Each leaf node has been labelled with the index i of the suffix $X[i..n-1]$ that it represents, where $i \in [0,n)$. All other explicit nodes are labelled V and root node r is outlined in bold. Suffix links are shown as dashed directed edges.

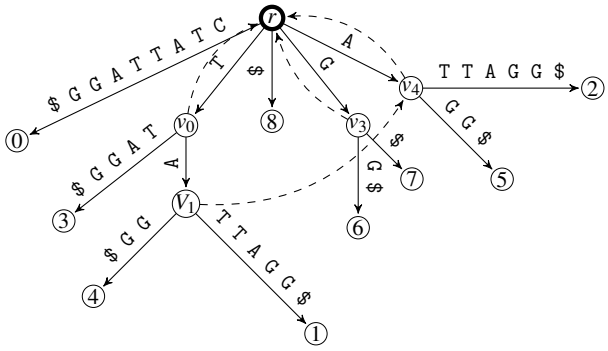
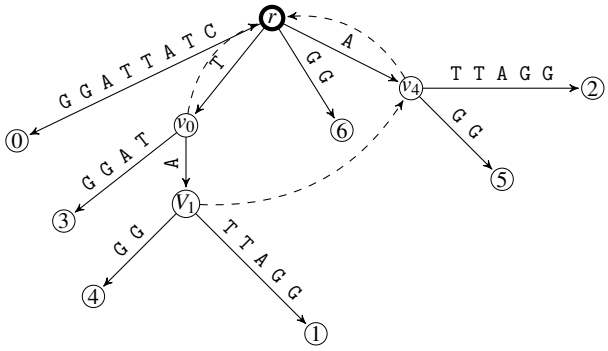


Figure 1.2 The implicit suffix tree of the string $X = \texttt{CTATTAGG}$. Each leaf node has been labelled with the index i of the suffix $X[i..n-1]$ that it represents, for all suffixes $i \in [0,7)$. All other explicit nodes are labelled V and root node r is outlined in bold. Suffix links are shown as dashed directed edges.



An *internal* node of ST_X is an explicit, non-root, non-leaf node. The *explicit child* of a node V, denoted as $\mathsf{explicitChild}(V)$, is the nearest explicit node that has an incoming edge from V. We define an *explicit parent* in a similar way. The *explicit ancestor* of a node U, denoted as $\mathsf{ancestor}(U)$, is any internal node that is an ancestor of U. The *explicit descendant* of an internal node V, denoted as $\mathsf{descendant}(V)$, is any internal node that is a descendant of V.

A *generalised suffix tree* $\mathsf{ST}_{X_0,\$_0,X_1,\$_1,..,X_{k-1},\$_{k-1}}$ is a suffix tree built from a set of k concatenated strings $\{X_0, X_1, .., X_{k-1}\}$, each separated by a unique symbol $\$_i \notin \{X_0, X_1, .., X_{k-1}\}$, where $i \in [0, k)$.

An in-depth discussion of suffix trees and their myriad uses can be found in [73].

Sequence Alignment Algorithms

Sequence alignment entails the structured comparison of sequences in order to deduce homology.

The Needleman-Wunsch pairwise sequence alignment algorithm [117] uses a dynamic programming approach to obtain the optimal *global* alignment of a pair of sequences of length n and m. The optimal global alignment corresponds to the lowest edit distance, which is also known as the Levenshtein distance [102] and is defined as the number of mutation, insertion or deletion operations required to transform one string in to the other. Insertions and deletions are collectively known as *indels*. The Needleman-Wunsch algorithm requires the construction of a matrix of size $\mathscr{O}(nm)$. Each cell in the matrix is filled in constant time by computing the best score from the following:

- the score in the cell diagonal to the current cell, plus the (penalty) score for a (mis)match; or
- the score in the cell above, or to the left of, the current cell, plus the penalty score for an indel.

Once the matrix is filled, the score in the bottom right corner represents the best score for the entire alignment. There may be multiple paths that could be traced from the bottom-right cell to the top-left cell, representing multiple alignments with the same score. The overall time complexity of the algorithm is $\mathscr{O}(nm)$.

Example 2. *Given the sequences $X = \texttt{GATCGCATT}$ and $Y = \texttt{GCTCGAGAC}$, the Needleman-Wunsch algorithm would return the following alignment:*

$$\texttt{G A T C G - - - C A T T}$$
$$\texttt{G C T C G A G A C - - -}$$

On the other hand, the Smith-Waterman algorithm [161] finds the best *local* alignment by tracing back the path from the highest scoring cell in the matrix, not necessarily the bottom-right cell. The traceback terminates at a cell with the value 0, which is the lowest score allowed in the Smith-Waterman scoring scheme. Thus, the algorithm finds the region which matches with the highest score in the pair of sequences, also in $\mathcal{O}(nm)$ time.

Example 3. *Given the sequences $X = \texttt{GATCGCATT}$ and $Y = \texttt{GCTCGAGAC}$ from Example 2, the Smith-Waterman algorithm would return the following alignment:*

$$\texttt{G A T C G}$$
$$\texttt{G C T C G}$$

Multiple sequence alignments (MSAs) are commonly constructed for phylogenetic analyses and progressive MSA is the most common method for their construction. First introduced in [55], the basic process of progressive MSA has three stages:

1. Pairwise sequence alignment of all pairs of sequences in the set.
2. Clustering sequences based on their similarity scores to build a guide tree.
3. Using the tree to guide the progressive addition of each sequence to the MSA.

Example 4. *Given the sequences* $X = \texttt{GATCGCATT}$ *and* $Y = \texttt{GCTCGAGAC}$ *from Example 3, and additional sequences* $A = \texttt{GGTCGTAGC}$ *and* $B = \texttt{GGTCGTTGC}$*, the following multiple sequence alignment would be produced, where conserved nucleotides are underlined:*

$$
\begin{aligned}
X &= \underline{G}\ A\ \underline{T\ C\ G}\ C\ A\ T\ T \\
Y &= \underline{G}\ C\ \underline{T\ C\ G}\ A\ G\ A\ C \\
A &= \underline{G}\ G\ \underline{T\ C\ G}\ T\ A\ G\ C \\
B &= \underline{G}\ G\ \underline{T\ C\ G}\ T\ T\ G\ C
\end{aligned}
$$

Figure 1.3 illustrates the resultant phylogenetic tree.

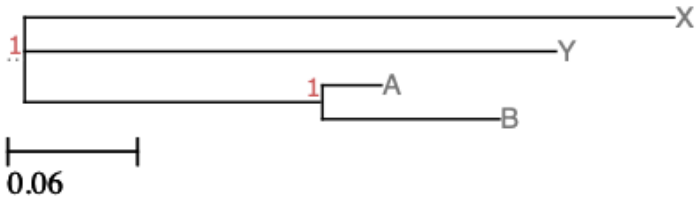


Figure 1.3 Each leaf of the tree is labelled with the name of the sequence it represents. Horizontal branch length represents the number of substitutions per base, and thus the evolutionary distance between a leaf node and its ancestor. Labels (in red) at branch splits represent the reliability of a split, where a value of 1 represents 100% reliability.

Note that all of the above-mentioned algorithms can take both nucleotide and amino acid sequences as input.

Chapter 2

Circular Sequence Comparison

2.1 Introduction

This chapter is based on work published in conference proceedings [68] and a journal [69], and has since been cited in 15 publications (see Section 2.6). The author's personal contribution to this work was in the experimentation.

2.1.1 Biological Motivation

Circular molecular structures are present, in abundance, in all domains of life: bacteria, archaea, and eukaryotes; and in viruses. They can be composed of either amino or nucleic acids. The following is an overview of such occurrences, and exhaustive reviews can be found in [41] for proteins and [75] for DNA.

Bacteria

Double-stranded, circular chromosomes and plasmids are found in most bacteria and archaea. Whole-genome comparison is a very useful tool in classifying bacterial strains, as well as

inferring phylogenetic associations between them. This is due to the dense structure of bacterial chromosomes, caused by the absence of introns, and the organisation of genes into operons. The additional benefit of aligning plasmids is the ability to identify important genes, such as antibiotic resistance genes, thereby enabling their study and exploitation by genetic engineering techniques [48].

Plants and Animals

The most familiar examples of such structures in eukaryotes are mitochondrial DNA (*MtDNA*). MtDNA is, in most cases, inherited solely from the mother, and thus it is generally conserved. Human MtDNA is double-stranded, with a length of 16,569 base pairs (bp), consisting of just 37 genes encoding 13 proteins and 24 RNA molecules [164]. The absence of recombination in these sequences allows them to be used as simple indicators of phylogenetic evolution [20, 150, 177], and their high mutation rate is a powerful discriminative feature [64, 174]. There also exist smaller structures, called extrachromosomal circular DNA, which are similar to plasmids in bacterial cells. They are described as one of the characteristics of genomic plasticity in eukaryotes [37] and may derive from MtDNA [97].

Viruses

It is common knowledge that many viral genomes are circular. Viral genomes vary greatly in size and structure. They can be made up of either RNA or DNA, and can be single- or double-stranded. Multiple sequence alignment (MSA, defined in Section 1.3.2) of viral genomes can be useful in the elucidation of novel sites of interest [23], as well as the inference of evolutionary relationships [21]. This is particularly important in studying the pathogenicity of viruses, due to the rapid rate of mutation of their genomes. Viroids are plant pathogens that contain very small, single-stranded, circular RNA. Their MSA could prove useful in the

analysis of their secondary structures and, therefore, the mechanisms by which viroids infect host plant cells [112].

Proteins

Naturally-occurring circular proteins are found in both prokaryotes and eukaryotes [41]. Bacteriocins are very small toxins produced by bacteria in order to compete with closely-related bacterial strains. Many of these are circular, including gassericin A, found in *Lactobacillus gasseri* LA39 [89], and circularin A, found in *Clostridium beijerinckii* [91]. An interesting phenomenon known to occur naturally in linear protein structures is circular permutation [178]. This is exemplified by swaposins: proteins highly-similar to saposins, resulting from circularly permuted linear peptide sequences [135]. The ability to align linear sequences from circular proteins can significantly speed up and enhance their analyses, and could also lead to the discovery of novel pairs of circularly permuted proteins.

2.1.2 Previous Work

Conventional tools, designed for linear sequences, could yield an incorrectly high genetic distance between closely-related circular sequences. Indeed, when sequencing molecules, the position where a circular sequence starts can be totally arbitrary. Due to this arbitrariness, a suitable rotation of one sequence would give much better results for a pairwise alignment (defined in Section 1.3.2), and hence highlight a similarity that any linear alignment would miss. A practical example of the benefit this can bring to sequence analysis is the following.

Example 5. *Linearized human (NC_001807) and chimpanzee (NC_001643) MtDNA sequences, obtained from GenBank [18], do not start in the same region. Their pairwise sequence alignment using EMBOSS Needle [139] (default parameters) gives a similarity of*

85.1% and consists of 1,195 gaps. However, taking different rotations of these sequences into account yields a much more significant alignment with a similarity of 91% and only 77 gaps.

Example 5 motivates the design of efficient algorithms that are specifically devoted to the comparison of circular sequences [8, 16, 17].

In this chapter, we consider the pairwise circular sequence comparison problem. Under the edit distance model, it consists in finding an optimal linear alignment of two circular strings X and Y, of length m and $n \geq m$ (without loss of generality), respectively. We consider the edit distance rather than the Hamming distance as it is more realistic in the context of molecular biology, where gaps representing insertion and deletion mutations must be allowed. Taking into account edit distance rather than Hamming distance is computationally challenging as the search space for seeking similarity is wider. The remainder of this section provides an overview of existing solutions for this problem.

Cubic Time Complexity

Trivially, the pairwise circular sequence comparison problem can be solved naïvely in $\mathcal{O}(nm^2)$ time, by using the Needleman-Wunsch algorithm (defined in Section 1.3.2) on the pair (X, Y_i) of sequences, for all $i < m$, where Y_i is the i^{th} rotation of Y. Hereafter, we refer to this approach as cyclical Needleman-Wunsch and denote it by cNW.

Super-Quadratic Time Complexity

In [106], the author presents a $\mathcal{O}(mn \log n)$-time solution using a divide-and-conquer approach, where the problem is reduced to finding the optimal path in the edit graph of strings X and YY. The edit graph is a directed acyclic graph, where horizontal and vertical edges represent deletions and insertions, respectively; and diagonal edges represent equivalence or

substitution. Any path in an edit graph defines the edit operations required to transform one sequence in to the other. Concatenating Y with itself ensures all rotations of Y are considered against X.

Lemma 1 ([106]). *Let j, k and l be three integers such that $0 \leq j < k < l \leq n$ and let $P(j)$ and $P(l)$ be two non-crossing paths in the edit graph of X and YY. Then there exists an optimal path $P(k)$ which does not cross either $P(j)$ or $P(l)$.*

The algorithm begins by computing $P(0)$, which is equivalent to $P(n-1)$. It then computes $P(\lceil \frac{n-1}{2} \rceil)$, where the search space is delimited by $P(0)$ and $P(n-1)$. Given Lemma 1, observe that the subsequent computation of $P(\lceil \frac{n-1}{4} \rceil)$ and $P(\lceil \frac{3n-1}{4} \rceil)$ are bounded by $\mathcal{O}(mn)$, and so on. Consequently, there are $\mathcal{O}(\log n)$ such computations, resulting in an overall time complexity of $\mathcal{O}(mn \log n)$.

Several other super-quadratic solutions also exist [109].

Other Solutions

Note that there are several approximate solutions to the problem that will not be mentioned in detail, for example the quadratic-time algorithm presented in [25]. Furthermore, this problem has also been considered under the Hamming distance model.

Application to Multiple Sequence Alignment (MSA)

A direct application of pairwise circular sequence comparison is progressive multiple circular sequence alignment [15, 113].

MSA: Using the Best Local Alignment. As well as implementing the naïve and exact algorithm, cNW, the cyclope [113] package contains an $\mathcal{O}(nm)$-time heuristic algorithm.

This algorithm first finds the best local alignment between two sequences, and then uses this as an anchor to rotate and align the remaining sequences iteratively, refining the anchor at each iteration. These pairwise alignment algorithms are therefore used to compute similarity scores for all pairs of sequences, and then produce an MSA.

MSA: Generalised Cyclic Suffix Tree. In [56], the authors propose the use of a data structure they term a *generalised cyclic suffix tree* (denoted as gcST) to find the best rotation of each sequence in a set of multiple sequences. Therefore, this can also be used for pairwise alignment, and works differently to progressive MSA algorithms. The gcST is constructed from all rotations of all sequences in the input set using concepts of Ukkonen's suffix tree construction algorithm [170], with modifications as follows. Suffix links on leaves act as connections between successive rotations of a string. Secondly, all leaves have the same depth of n. Finally, each node is augmented with a bit-vector indicating in which sequences of the set its corresponding factor occurs. The gcST is then used in the following way:

Step 1 Nodes are identified that represent factors that are common to all sequences using depth-first search.

Step 2 Nodes that represent suffixes of such common factors are discarded (making use of suffix links), resulting in a set of maximal common factors.

Step 3 Nodes that represent factors repeated in the same sequence are discarded, resulting in a set of unique maximal common factors.

Step 4 Nodes are clustered such that each cluster represents a sequence of factors occurring consecutively and in the same order in each sequence in the set, where the maximum allowed gap between each factor is 10bp.

Step 5 The longest such chain of factors is used to determine the corrected rotation of each sequence.

This algorithm was implemented as tool CSA, which unfortunately, is no longer maintained. Notably, it restricts the number of sequences in the set to be 32 and requires there to be at least one factor that occurs in every sequence only once.

MSA: Other Solutions. Multiple circular sequence alignment has also been considered in [99] under the Hamming distance model.

Filtering Techniques

Algorithms that speed up the process of string matching, by filtering out candidate positions in which a particular string can never occur, are known as *filters*. Filters that work for Hamming distance do not work in general for edit distance [130] as well. An exception to this are the q-gram filtering techniques [169] that have successfully been used for string matching under the edit distance model (e.g. [27, 124, 138]), as well as for multiple local alignments, both under the Hamming [123] and edit [124] distance models.

2.1.3 Chapter Summary

To the best of our knowledge, there is no fast (that is, with sub-quadratic time complexity) and exact (or at least very accurate) algorithm for circular sequence comparison under some realistic model (that is, allowing insertions and deletions). Here we present new efficient q-gram-based methods for pairwise circular sequence comparison. Specifically, our contribution is threefold.

Distance pseudometric. In Section 2.2, we introduce the *β-blockwise q-gram distance* between two strings X and Y, that is, a more powerful generalisation of the q-gram string distance introduced in [169]. Intuitively, and similarly to [27, 124, 138], this generalisation comprises partitioning X and Y in β *blocks* each, as evenly as possible,

computing the q-gram distance between the corresponding block pairs, and then summing up the distances computed blockwise.

Algorithm. In Section 2.3, we present an algorithm based on the suffix array [108] that finds the rotation of X such that the β-blockwise q-gram distance between the rotated X and Y is minimal, in $\mathcal{O}(\beta m+n)$ time and space, where $m=|X|$ and $n=|Y|$, thereby *exactly* solving the circular sequence comparison problem under the β-blockwise q-gram distance measure. We also present a simple heuristic algorithm to solve an *approximate* version of the problem.

Experimental results. In Section 2.5, we present an experimental study, using real and synthetic data, which demonstrates orders-of-magnitude superiority of our approach, in terms of efficiency, while maintaining an accuracy very competitive to the *optimal* obtained after considering all rotations of X against Y using cNW.

2.2 Definitions

We refer to any string $X \in \Sigma^q$ as a *q-gram*. Two strings are considered k-abelian equivalent for some positive integer k, if they have the same length and share the same factors of length at most k, including multiplicities. Note that if k is greater than or equal to the string's length, then the strings must be equal. A version of this result, called extended k-abelian equivalence, focuses only on the factors of length k [51]. By setting $k=q$, it is quite straightforward to notice the equivalence with q-grams. Therefore, in order to avoid confusion we will refer to the former notion from now on as *q-abelian equivalence*.

The *Parikh vector* associated with a string $Z \in \Sigma^*$ is denoted by $\mathcal{P}(Z)$ and represents a vector of size $|\Sigma|$, where each component denotes the number of occurrences in Z of the corresponding letter from Σ.

A circular string, of length m, can be viewed as a traditional linear string which has the left- and right-most letters wrapped around and glued together in some way. Under this notion, the same circular string can be seen as m linear strings, each of length m. Given a string X of length m, we denote by $X^i = X[i..m-1]X[0..i-1]$, where $i \in (0,m)$, the ith *rotation* of X; and $X^0 = X$.

Example 6. *The string* $X = X^0 =$ `abababbc` *has the following rotations:* $X^1 =$ `bababbca`, $X^2 =$ `ababbcab`, $X^3 =$ `babbcaba`, $X^4 =$ `abbcabab`, $X^5 =$ `bbcababa`, $X^6 =$ `bcababab`, $X^7 =$ `cababab`.

We give some further definitions following [169]. The *q-gram profile* of a string X is the vector $G_q(X)$, where $q > 0$ and $G_q(X)[V]$ denotes the total number of occurrences of q-gram $V \in \Sigma^q$ in X. The q-gram distance between two strings X and Y is defined as

$$D_q(X,Y) = \sum_{V \in \Sigma^q} \left| G_q(X)[V] - G_q(Y)[V] \right|. \tag{2.1}$$

Note that D_q is a *pseudo-metric*, as $D_q(X,Y)$ can be 0 even if $X \neq Y$. Its properties can be found in [169].

Example 7. *Let* $X =$ `GGAGTCTA`, $Y =$ `TTCTAGCG`, *and* $q = 3$. *Table 2.1 shows the q-gram profiles of strings X and Y and the q-gram distance* $D_q(X,Y) = 8$ *between them. Each row represents the frequency of a q-gram in the given string. For succinctness of presentation, only those rows with frequency greater than zero (in either string) are shown, as well as rows representing* `AAA`, `CCC`, `GGG`, *and* `TTT` *as points of reference.*

For a given integer parameter $\beta \geq 1$, we define a generalization of the q-gram distance in Equation (2.1) by partitioning X and Y in β *blocks* as evenly as possible, and computing the

(a)	$G_q(X)$	**(b)**	$G_q(Y)$	**(c)**	$D_q(X,Y)$
AAA	0	AAA	0	AAA	0
AGC	0	AGC	1	AGC	1
AGT	1	AGT	0	AGT	1
CCC	0	CCC	0	CCC	0
CTA	1	CTA	1	CTA	0
GAG	1	GAG	0	GAG	1
GCG	0	GCG	1	GCG	1
GGA	1	GGA	0	GGA	1
GGG	0	GGG	0	GGG	0
GTC	1	GTC	0	GTC	1
TAG	0	TAG	1	TAG	1
TCT	1	TCT	1	TCT	0
TTC	0	TTC	1	TTC	1
TTT	0	TTT	0	TTT	0

Table 2.1 The q-gram profiles of strings X and Y from Example 7 and the q-gram distance between them.

q-gram distance between each pair of blocks, one from X and one from Y. The rationale is to enforce *locality* in the resulting overall distance. For the sake of presentation in the rest of the chapter, we assume that the lengths $|X| = m$ and $|Y| = n$ are both multiples of β, so that X and Y are conceptually partitioned into β blocks, each of size m/β for X and n/β for Y. Setting an appropriate value for β in practice is discussed in Section 2.5.

Definition 1. *Given strings X, of length m, and Y, of length $n \geq m$, and integers $\beta \geq 1$ and $q > 0$, the* β-blockwise q-gram distance $D_{\beta,q}(X,Y)$ *is defined as*

$$D_{\beta,q}(X,Y) = \sum_{j=0}^{\beta-1} D_q\left(X\left[\frac{jm}{\beta}..\frac{(j+1)m}{\beta}-1\right], Y\left[\frac{jn}{\beta}..\frac{(j+1)n}{\beta}-1\right]\right). \tag{2.2}$$

Example 8. *Following Example 7, let $X =$ `GGAGTCTA` and $Y =$ `TTCTAGCG`, $q = 3$, and $\beta = 2$. Further let $X_1 =$ `GGAG`, $X_2 =$ `TCTA` and $Y_1 =$ `TTCT`, $Y_2 =$ `AGCG` be the two blocks of X and*

Y, respectively. Table 2.2 shows the q-gram profiles of strings X_1, X_2, Y_1, and Y_2; and the q-gram distance between X_1 and Y_1 and the q-gram distance between X_2 and Y_2.

(a) $G_q(X_1)$

AAA	0
AGC	0
AGT	0
CCC	0
CTA	0
GAG	1
GCG	0
GGA	1
GGG	0
GTC	0
TAG	0
TCT	0
TTC	0
TTT	0

(b) $G_q(Y_1)$

AAA	0
AGC	0
AGT	0
CCC	0
CTA	0
GAG	0
GCG	0
GGA	0
GGG	0
GTC	0
TAG	0
TCT	1
TTC	1
TTT	0

(c) $D_q(X_1,Y_1)$

AAA	0
AGC	0
AGT	0
CCC	0
CTA	0
GAG	1
GCG	0
GGA	1
GGG	0
GTC	0
TAG	0
TCT	1
TTC	1
TTT	0

(d) $G_q(X_2)$

AAA	0
AGC	0
AGT	0
CCC	0
CTA	1
GAG	0
GCG	0
GGA	0
GGG	0
GTC	0
TAG	0
TCT	1
TTC	0
TTT	0

(e) $G_q(Y_2)$

AAA	0
AGC	1
AGT	0
CCC	0
CTA	0
GAG	0
GCG	1
GGA	0
GGG	0
GTC	0
TAG	0
TCT	0
TTC	0
TTT	0

(f) $D_q(X_2,Y_2)$

AAA	0
AGC	1
AGT	0
CCC	0
CTA	1
GAG	0
GCG	1
GGA	0
GGG	0
GTC	0
TAG	0
TCT	1
TTC	0
TTT	0

Table 2.2 The q-gram profiles of strings X_1, X_2, Y_1, and Y_2; the q-gram distance $D_q(X_1,Y_1) = 4$ between X_1 and Y_1; and the q-gram distance $D_q(X_2,Y_2) = 4$ between X_2 and Y_2, giving $D_{\beta,q}(X,Y) = D_q(X_1,Y_1) + D_q(X_2,Y_2) = 4+4 = 8$.

In this paper, we consider the following problem, where we search for the i^{th} rotation of X that minimizes its blockwise distance from Y as defined in (2.2). Ties are broken arbitrarily.

CIRCULAR SEQUENCE COMPARISON (CSC)

Input: Strings X and Y of lengths m and $n \geq m$, respectively, and integers $\beta \geq 1$ and $q < m$.

Output: Integer i such that $D_{\beta,q}(X^i,Y)$ is minimal.

2.3 Algorithms

We use the following result to first give a naïve solution to the CSC problem.

Lemma 2 ([169]). *If we have space $\mathcal{O}(|\Sigma|^q)$ available, then the q-gram distance $D_q(X,Y)$ can be computed in time $\mathcal{O}(m+n)$ and extra space $\mathcal{O}(m+n)$, where $m = |X|$ and $n = |Y|$.*

We then apply Lemma 2 to each pair of blocks of X and Y separately.

Lemma 3. *If there is space $\mathcal{O}(|\Sigma|^q)$ available, then the β-blockwise q-gram distance $D_{\beta,q}(X,Y)$ can be computed in time $\mathcal{O}(m+n)$ and extra space $\mathcal{O}(\frac{m+n}{\beta})$, where $m = |X|$ and $n = |Y|$.*

Recall that one of the two input strings should be concatenated with itself to ensure that all rotations are represented. The naïve algorithm, denoted by nCSC, computes for $X' = XX$ the values

$$\delta_i = D_{\beta,q}(X'[i..i+m-1], Y),$$

for all $i \in [0,m)$; we report position i such that δ_i is minimal. This requires the application of Lemma 3, m times. Therefore, we obtain the following.

Lemma 4. *If we have space $\mathcal{O}(|\Sigma|^q)$ available, then algorithm nCSC solves the CSC problem in time $\mathcal{O}(m(m+n))$ and extra space $\mathcal{O}(\frac{m+n}{\beta})$.*

2.3.1 Algorithm hCSC: a Heuristic Algorithm

Here we give a simple heuristic algorithm, denoted by hCSC, to solve the CSC problem faster than nCSC, and return an approximation of the best rotation.

Step 1: We split $X' = XX$ in 2β non-overlapping string *blocks* of length m/β. We obtain strings $X_0, X_1, \ldots, X_{2\beta-1}$, such that

$$X_i = X'[\frac{im}{\beta} .. \frac{(i+1)m}{\beta} - 1] \ \ \forall \ \ i \in [0, 2\beta).$$

We split Y in β non-overlapping string blocks of length n/β. We obtain strings $Y_0, Y_1, \ldots, Y_{\beta-1}$, such that

$$Y_i = Y[\frac{in}{\beta} .. \frac{(i+1)n}{\beta} - 1] \ \ \forall \ \ i \in [0, \beta).$$

Step 2: For a given sequence $X_j, \ldots, X_{j+\beta-1}$ of strings and Y, we compute the β-blockwise q-gram distance as follows

$$\delta_j = D_{\beta,q}(X'[\frac{jm}{\beta} .. \frac{(j+1)m}{\beta} - 1], Y) = \sum_{i=0}^{\beta-1} D_q(X_{j+i}, Y_i) \ \ \forall \ \ j \in [0, \beta].$$

We choose $j_{best} = j$ such that δ_j is minimal, for all $j \in [0, \beta]$. In other words, we have found a *window* of length m starting at position j_{best}, such that

$$(j_{best} + 1) \bmod (m/\beta) = 0,$$

consisting of β blocks of length m/β each, that minimizes its β-blockwise q-gram distance from y.

Step 3 To perform a refinement on the position of the window, we consider all starting positions included in the two blocks starting at positions j_{best} and $j_{best} - m/\beta$. This includes $2m/\beta - 1$ starting positions in total—we do not need to consider position $j_{best} - m/\beta$ as this was already considered by another window in Step 2. Similarly to Step 2, we obtain the β-blockwise q-gram distance δ_i between the window starting at position i of X' and Y, for all $i \in (j_{best} - m/\beta, j_{best} + m/\beta)$. We report position $i_{best} = i$ such that δ_i is minimal, for all $i \in (j_{best} - m/\beta, j_{best} + m/\beta)$.

Analysis

Step 1 can be done trivially in $\mathcal{O}(m+n)$ time. If we have $\mathcal{O}(|\Sigma|^q)$ space available, then, by Lemma 2, $D_q(X_{j+i}, Y_i)$ can be computed in $\mathcal{O}(\frac{m+n}{\beta})$ time. By Lemma 3, δ_j can be computed in $\mathcal{O}(\beta(\frac{m+n}{\beta})) = \mathcal{O}(m+n)$ time. Hence, Step 2 can be done in $\mathcal{O}(\beta(m+n))$ time. In Step 3, the blockwise q-gram distance δ_i between a single window and Y can be computed in $\mathcal{O}(\beta(\frac{m+n}{\beta})) = \mathcal{O}(m+n)$ time. There exist $2m/\beta - 1$ such windows. Hence, Step 3 can be done in $\mathcal{O}(\frac{m(m+n)}{\beta})$ time. Overall, the algorithm requires $\mathcal{O}(\beta(m+n) + \frac{m(m+n)}{\beta})$ time and $\mathcal{O}(|\Sigma|^q + m + n)$ space.

For practical purposes, setting $\beta = \mathcal{O}(\sqrt{m})$ and $q = \mathcal{O}(\log_{|\Sigma|} m)$ gives an algorithm with time complexity $\mathcal{O}(\sqrt{m}(m+n))$ and space complexity $\mathcal{O}(m+n)$.

2.3.2 Algorithm saCSC: an Exact Suffix-Array-Based Algorithm

The heuristic hCSC does not guarantee to find the exact value i, for which $\delta_i = D_{\beta,q}(X^i, Y)$ is minimal. In particular, when we identify j_{best} in Step 2, that is, a j for which δ_j is minimal, we take into account only the values of j such that $(j+1) \bmod (m/\beta) = 0$. Thus, Step 3 cannot guarantee that i_{best}, the local minimum obtained by shifting the window m/β positions to the right and left of j_{best}, is minimal for all $i \in [0, m)$. In this section, we give a fast and exact algorithm, denoted by saCSC, to find i such that $\delta_i = D_{\beta,q}(x^i, y)$ is minimal, based on the suffix array (defined in Section 1.3.2).

We partially follow the idea from [51]. The authors propose a linear-time algorithm to solve the string matching problem when looking at q-abelian equivalent strings: given a string X of length m, a string Y of length $n \geq m$, and a positive integer $q < m$, all factors of Y that are q-abelian equivalent to X can be found in $\mathcal{O}(m+n)$ time and space. The idea of the algorithm in [51] consists of constructing the suffix array of the string XY, and ranking sets

of identical q-length prefixes of suffixes in the suffix array in the order of their appearance. Then it constructs new strings based on this ranking, and solves the problem as in the *jumbled matching* case [26]; that is, identifying all factors of Y that have the same Parikh vector as X.

We first describe our algorithm for a single block ($\beta = 1$) and then address the general case ($\beta \geq 1$).

Basic Algorithm for $\beta = 1$

We construct the suffix array of the string $Z = XXY$ and assign a *rank* to the prefix, of length q, of each suffix, of length at least q, based on its order in the suffix array. That is, the first i_0 suffixes, of length at least q, in the suffix array, all sharing the same prefix of length q, will get rank 0; the next i_1 suffixes, of length at least q, sharing the same prefix of length q, different from the previous one, will get rank 1; and so on. Next, based on this ranking, we construct two new strings X', of length $2m-q+1$, and Y', of length $n-q+1$, such that $X'[i] = j$, if j is the rank of the q-length prefix of the $(i+1)$th suffix of XX in the suffix array of Z; the same goes for Y. It is not difficult to see that the ranks go up at most to value $m+n-q+1$. However, we can reduce this value to $m+2$ by introducing two new ranks a_X and a_Y: we can conceptually replace every letter of X' that does not occur in Y' by a_X, and every letter of Y' that does not occur in X' by a_Y. Hence we can consider that the new strings X' and Y' are defined over an integer alphabet of size *at most* $\min(n-q+1, m)+2 \leq m+2$.

Example 9. *Let $X =$ `GAGTCTA`, $Y =$ `TCTAGCG`, $q = 3$ and $Z = XXY$. The table below shows the suffix array SA and LCP array LCP of Z, as well as strings X' and Y'. Observe that, $X'[3] = Y'[0] = 2$ denotes that $X[3..5] = Y[0..2] =$ `TCT` and $X'[0] = a_X$ denotes that $X[0..2] =$ `GAG` does not occur in Y.*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$Z[i]$	G	A	G	T	C	T	A	G	A	G	T	C	T	A	T	C	T	A	G	C	G
$SA[i]$	6	17	1	8	13	19	4	15	11	20	0	7	18	2	9	5	16	12	3	14	10
$LCP[i]$	0	2	2	6	1	0	1	4	3	0	1	7	1	1	5	0	3	2	1	5	4
$X'[i]$	a_X	a_X	a_X	2	0	1	a_X	a_X	a_X	a_X	2	0									
$Y'[i]$	2	0	1	a_Y	a_Y																

We observe that when identifying the q-gram distance between two blocks, we can apply the idea in [51], with the only difference that we should also maintain a Parikh vector that stores the *differences* between the number of occurrences of q-grams in the current block of XX and Y. Recall that the new alphabet of ranks, upon which X' and Y' are built, represent q-grams occurring in X and Y.

At the time of construction of Y', we also construct a Parikh vector $\mathscr{P}(Y')$, which stores the number of occurrences of each letter in Y'. Notice that $|\mathscr{P}(Y')| \leq m+2$. Later on, when computing the q-gram distances, we can construct another vector diff to store the letter differences between $\mathscr{P}(Y')$ and the Parikh vector covering the $m-q+1$ letters of X' associated with a window of length m on the string XX. This gives us the current Parikh difference and, in fact, represents the q-gram distance between the two analysed blocks, where $|\mathsf{diff}| \leq m+2$. Apart from these, we only need another vector δ of size m, which stores at each position i the actual q-gram distance δ_i between Y and the window starting at position i in XX, which is the ith rotation X^i of X.

We use a sliding window of length m to maintain the above information. When the window is shifted one position to the right, we have to update the vector diff by incrementing the value for the first letter of the previous window, and decrementing the value for the last letter of the

current window. The distance δ_i between Y' and the factor of X' starting at position i is thus updated using the value of the q-gram distance δ_{i-1} as follows. After adding the first letter of the previous window to the vector diff, if we have a non-positive value for this letter, we update the distance δ_i by decreasing the previous value δ_{i-1} by 1; otherwise, we increase it by 1. If, after decrementing the value in diff for the last letter of the current window, we have a non-negative value, we update the distance δ_i by decreasing the previous value by 1; otherwise, we increase it by 1. The distance δ_i will never be less than the number of occurrences of a_Y. Furthermore, if the first letter of the previous window was a_X, the new distance decreases by 1; and for every newly added a_X, it increases by 1. As these operations require constant time, after going through X' with Y' once, we obtain the list of distances δ_i from Y to each rotation X^i in linear time.

We are now able to give a more formal description of the steps to solve the CSC problem for $\beta = 1$.

Step 1: Construct the SA, iSA, and LCP of $Z = XXY$. Rank the q-length prefixes of suffixes using LCP array queries. Construct X' and Y', as well as $\mathscr{P}(Y')$, the Parikh vector storing, for each letter of Y', the number of its occurrences in Y'; making proper use of letters a_X and a_Y, respectively. Further, initialise diff $= \mathscr{P}(Y')$ and $\delta_0 = \sum_{i=0}^{|\mathscr{P}(Y')|-1} \mathscr{P}(Y')[i]$.

Example 10. *Following Example 9, let* $X =$ `GAGTCTA`, $Y =$ `TCTAGCG`, $q = 3$, *and* $Z = XXY$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$Z[i]$	G	A	G	T	C	T	A	G	A	G	T	C	T	A	T	C	T	A	G	C	G
$X'[i]$	a_X	a_X	a_X	2	0	1	a_X	a_X	a_X	a_X	2	0									
$Y'[i]$	2	0	1	a_Y	a_Y																

The table below represents vector diff, right after the execution of Step 1. By summing the values in the table, we observe that $\delta_0 = 5$.

a_X	0
a_Y	2
0	1
1	1
2	1

Step 2: Read the first $m-q+1$ letters of X', which constitute our sliding window of length m on the string XX. When reading letter $X'[i]$, update diff by decreasing the value for $X'[i]$ by one, and update δ_0 accordingly, as follows:

$$\mathsf{diff}[X'[i]] = \mathsf{diff}[X'[i]] - 1 \qquad \text{and} \qquad \delta_0 = \begin{cases} \delta_0 - 1, & \text{if } \mathsf{diff}[X'[i]] \geq 0 \\ \delta_0 + 1, & \text{if } \mathsf{diff}[X'[i]] < 0. \end{cases}$$

Example 11. *Following Example 10, the table below represents vector diff, right after the execution of Step 2. By summing the values in the table, we observe that* $\delta_0 = 6$.

a_X	3
a_Y	2
0	0
1	1
2	0

Step 3: Let i be the current position in X' and repeat this step, one position at a time. Shift the window to the right, update the information for diff

$$\mathsf{diff}[X'[i]] = \mathsf{diff}[X'[i]] + 1 \qquad \text{and} \qquad \mathsf{diff}[X'[i+m]] = \mathsf{diff}[X'[i+m]] - 1,$$

and calculate δ_{i+1}, based on this information, sequentially applying the two following rules

$$\delta_{i+1} = \begin{cases} \delta_i - 1, & \text{if } \mathsf{diff}[X'[i]] \leq 0 \\ \delta_i + 1, & \text{if } \mathsf{diff}[X'[i]] > 0 \end{cases}$$

$$\delta_{i+1} = \begin{cases} \delta_{i+1} - 1, & \text{if } \mathsf{diff}[X'[i+m]] \geq 0 \\ \delta_{i+1} + 1, & \text{if } \mathsf{diff}[X'[i+m]] < 0. \end{cases}$$

Example 12. *Following Example 11, the table below represents vector* diff *at iteration* $i' = 3$ *of Step 3. By summing the values in the table, we observe that* $\delta_0 = 4$*. Therefore,* $X^3 = \texttt{TCTAGAG}$ *is in fact the best rotation of* X*, given* Y*.*

a_X	2
a_Y	2
0	0
1	0
2	0

Correctness

Steps 1 and 2 are trivially correct as at the end of them we have that diff is the difference between $\mathscr{P}(Y')$ and the Parikh vector corresponding to the window on X'. These operations follow directly from the definitions of SA and LCP, and are followed by a simple traversal of SA, in order to obtain the ranks, and create vectors $\mathscr{P}(Y')$ and diff. Recall that, δ_0 was initially the number of letters in Y', and note its behavior as follows:

- δ_0 is decreasing as long as the difference between the Parikh vectors for a specific letter is non-negative. Thus, we have more occurrences of that letter in Y' compared to the window on X'.

- δ_0 is increasing otherwise.

In Step 3, we update diff by incrementing $\mathsf{diff}[X'[i]]$ for the letter $X'[i]$ and decrementing $\mathsf{diff}[X'[i+m]]$ for the new letter $X'[i+m]$. The q-gram distance δ_i at that position is based on the updated values of diff, as well as the q-gram distance δ_{i-1} at the previous position. If $\mathsf{diff}[X'[i]] \leq 0$, then there were more letters $X'[i]$ in Y' than in the window, thus we need to decrease the distance. Alternatively, if $\mathsf{diff}[X'[i]] > 0$, then there were at least as many letters $X'[i]$ in the window as in Y', and taking one out increases the distance. Complementary reasoning applies to the newly added letter $X'[i+m]$.

Finally, note the following properties of δ_i:

- It never goes below the number of occurrences of a_Y in Y'.
- It is equal to the number of occurrences of a_Y in Y' when all other elements of diff are 0.
- It represents the q-gram distance between Y and X^i, the corresponding window of length m starting at position i in XX.

Analysis

In Step 1, constructing SA, iSA, and LCP of XXY can be done in time and extra space $\mathcal{O}(m+n)$ (Section 1.3.2). Furthermore, the construction of X', Y', $\mathcal{P}(Y')$, diff, and δ_0 is done with the same time and space cost. In Step 2, updating diff and δ_0 after reading each letter takes constant time, as we execute two operations, thus $\mathcal{O}(m)$ in total. Constant time is required for each iteration in Step 3 to compute the value of δ_i, where $i \in [1,m)$, and update diff, since a constant number of operations are executed, thus $\mathcal{O}(m)$ in total. Hence, we can solve the CSC problem for $\beta = 1$ in time and space $\mathcal{O}(m+n)$.

General Algorithm for $\beta \geq 1$

We can now generalise this algorithm to solve the CSC problem for any $\beta \geq 1$, which gives algorithm saCSC. We maintain a Parikh vector for each block, and apply the above basic algorithm for the jth block in each string, computing their q-gram distance. If we denote by $\mathscr{P}_j(Y')$ and diff_j, for all $j \in [0,\beta)$, the β Parikh vectors of Y' and of the q-gram distances, respectively, as well as by $\delta_{i,j}$ the q-gram distance between the jth block of Y and the jth block of X^i, then the updates will be given by the formulae shown below.

Specifically, when shifting the window one position to the right from position i, update the information for every diff_j, where $j \in [0,\beta)$, as follows

$$\mathsf{diff}_j[X'[i+\frac{jm}{\beta}]] = \mathsf{diff}_j[X'[i+\frac{jm}{\beta}]] + 1$$
$$\mathsf{diff}_j[X'[i+\frac{(j+1)m}{\beta}]] = \mathsf{diff}_j[X'[i+\frac{(j+1)m}{\beta}]] - 1,$$

and calculate $\delta_{i+1,j}$, based on this information, sequentially applying the two following rules

$$\delta_{i+1,j} = \begin{cases} \delta_{i,j} - 1, & \text{if } \mathsf{diff}_j[X'[i+\frac{jm}{\beta}]] \leq 0 \\ \delta_{i,j} + 1, & \text{if } \mathsf{diff}_j[X'[i+\frac{jm}{\beta}]] > 0 \end{cases}$$

$$\delta_{i+1,j} = \begin{cases} \delta_{i+1,j} - 1, & \text{if } \mathsf{diff}_j[X'[i+\frac{(j+1)m}{\beta}]] \geq 0 \\ \delta_{i+1,j} + 1, & \text{if } \mathsf{diff}_j[X'[i+\frac{(j+1)m}{\beta}]] < 0. \end{cases}$$

At each position $i < m$, we can update all of the β Parikh vectors corresponding to the blocks, as previously described, in time $\mathscr{O}(\beta)$. Therefore, we obtain the following result.

Theorem 1. *Algorithm saCSC solves the* CSC *problem in $\mathscr{O}(\beta m + n)$ time and space.*

2.4 Implementation

We implemented algorithms nCSC, hCSC, and saCSC as the program CSC, distributed under the GNU General Public License (GPL), and freely available at `http://github.com/solonas13/csc`. Given the following parameters, CSC finds the rotation of X (or an approximation of it) that minimizes its β-blockwise q-gram distance from Y.

- One of three algorithms: nCSC, hCSC, and saCSC.
- Two sequences X and Y in (Multi)FASTA format.
- The number β of blocks.
- The length q of the q-grams.

For comparison purposes, we also implemented the naïve algorithm. Recall that it compares all rotations of X against Y using the Needleman-Wunsch algorithm (defined in Section 1.3.2) with substitution matrices and affine gap penalty scores [65]; we denote this implementation by cNW. We also implemented the following heuristics. We first use the Smith-Waterman local alignment algorithm (defined in Section 1.3.2) to search for the best local alignment of X and Y and then use a central match from this local alignment to anchor the global alignment; we denote this implementation by hSW.

2.4.1 Refining Algorithm saCSC

The application of the β-blockwise q-gram distance via algorithm saCSC suggests that an optimal or a close-to-optimal rotation of X can be found when compared to cNW. Due to the locality property offered by the newly introduced distance notion, it is reasonable to assume

that a close-to-optimal rotation returned by saCSC may be refined via some quick heuristics that take into consideration the blocks at both ends.

Let X^i be a close-to-optimal rotation of X returned by saCSC. We introduce a new input parameter $p \in (0, \frac{\beta}{3}]$, which defines the length L of the prefixes and suffixes of X^i and Y to be considered in the refinement as follows:

$$L = \lfloor p \times \frac{m}{\beta} \rfloor.$$

We take p block(s) of the *prefix* of X^i, concatenate it with a string of equal length L comprised only of letter \$, where $\$ \notin \Sigma$, and concatenate that with p block(s) of the *suffix* of X^i to form a new string X'' of length $3L$. We do the same with Y to form a new string Y''. Setting a reasonable value for p in practice is explored in Table 2.5.

The refinement algorithm works by taking all rotations of X'' and comparing their similarity to Y''. Each rotation of X'' is compared to Y'' excluding when a \$ letter is found at index 0 of the rotation of X''. We measure the similarity between the strings for which equality between letters are positively valued; inequalities, insertions, and deletions are negatively valued; and comparisons involving \$ are neither positively nor negatively valued. The goal of rotating X'' serves to find the rotation that maximizes the similarity to Y'' and, to this end, we make use of the Needleman-Wunsch algorithm (defined in Section 1.3.2). The rotation of X'' which results in the maximum score is chosen as the best rotation, and hence, the final rotation $X^{i'}$ of X is computed based on this rotation of X''. Ties are broken arbitrarily. We denote this new algorithm, consisting of saCSC and the refinement stage, by saCSCr.

The application of the Needleman-Wunsch algorithm on strings of length $3L$ has a time complexity of $\mathcal{O}(L^2)$. Considering all rotations of X'' results in a time complexity of $\mathcal{O}(L^3)$ for the refinement step. Overall, saCSCr takes time $\mathcal{O}(\beta m + n + L^3)$.

Example 13. *Consider the following pair of strings obtained from* saCSC

$$X^i = \texttt{GACACCCCCCACAGTTTATGTAGCTT...ACCCCGAACCAACCAAACCCCAAA}$$

$$Y = \texttt{GTTTATGTAGCTTACCTCCCCAAAGC...CAAACCCCAAAGACACCCCACACA}$$

and the following pair of strings formed from the prefixes and suffixes of X^i *and* Y *for* $L = 25$.

$$X'' = \texttt{GACACCCCCCACAGTTTATGTAGCTT\$ACCCCGAACCAACCAAACCCCAAA}$$

$$Y'' = \texttt{GTTTATGTAGCTTACCTCCCCAAAG\$CAAACCCCAAAGACACCCCACACA}$$

The Needleman-Wunsch algorithm for all rotations of X'' *and string* Y'' *gives the following optimal alignment*

$$\texttt{GTTTATGTAGCTT\$ACCCCGAACCAACCAAACCCCAAAGACACCCCCCACA}$$

$$\texttt{GTTTATGTAGCTTACCTCCCCAAAG\$CAAACCCCAAAGACACCCCACACA}$$

which tells us that a refined rotation is in fact $X^{i'}$, *where* $i' = i - 13$

$$X^{i'} = \texttt{GTTTATGTAGCTT...CAAACCCCAAAGACACCCCCCACA}$$

$$Y = \texttt{GTTTATGTAGCTT...CAAACCCCAAAGACACCCCACACA}$$

2.5 Experimental Results

The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600 CPU at 3.4GHz and 12GB of RAM under 64-bit GNU/Linux. All programs were compiled with `gcc` version 4.7.3. We used both synthetic data and real data. All input datasets referred to in this section are publicly maintained at the same web-site. First, in

Sections 2.5.1–2.5.2, we establish the quality (accuracy and performance) of our methods. Then, in Sections 2.5.3–2.5.4, we show applications of our methods.

2.5.1 Accuracy

We began with simulating three DNA sequence datasets using INDELible [60], which produces linear sequences with substitutions, insertions, and deletions at rates defined by the user. Each dataset consisted of 12 sequences (denoted by α), each of length approximately 2,500 bp (denoted by γ). Three unique substitution rates (denoted by θ) were set, per dataset, using the substitution model JC69 (Jukes-Cantor, 69): 5%, 20%, and 35% (similar to those observed in MtDNA in mammals [115]). The insertion and deletion rates were set, respectively, to 4% and 6% (denoted by κ and ω), relative to substitution rate of 1 (similar to those observed in MtDNA in mammals [56]). We refer to these datasets as *Original*.

To allow for comparison of the performance of the algorithms in realigning randomly rotated sequences, which should be similar to those obtained from sequencing circular DNA structures, one random rotation was generated in each sequence in all datasets, creating new datasets which will be referred to as *Random*. Using the three *Random* datasets, consisting of 12 sequences each, allowed us to test the accuracy of hCSC and saCSC; notice that nCSC and saCSC always return the same rotation. For each *Random* dataset, an all-against-all sequence comparison was performed. That is, all $nCr = \frac{n!}{r!(n-r)!} = \frac{12!}{2!(12-2)!} = 66$ distinct pairs of sequences in each dataset were given as input to both hCSC and saCSC. Parameter β was set to $\lceil\sqrt{m}\rceil = 50$ and q was set to $\lceil\log_{|\Sigma|} m\rceil = 6$. The resultant re-rotated sequences were aligned using EMBOSS Needle and the similarity scores were compared to those of the *Original* and *Random* datasets, which were input directly to EMBOSS Needle. The results can be found in Figure 2.1. Note that EMBOSS Needle is an implementation of the

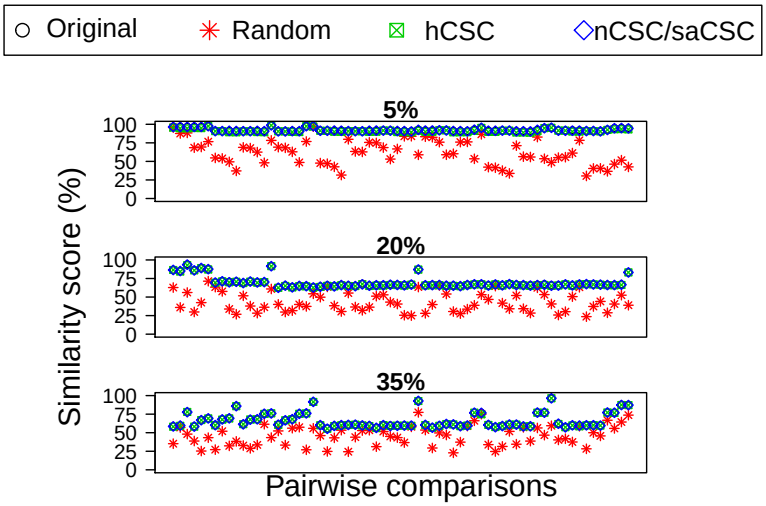


Figure 2.1 Accuracy comparison for substitution rates 5%, 20%, and 35%.

Needleman-Wunsch algorithm and any mention of it in this chapter implies the use of default parameters.

The results in Figure 2.1 show that:

- Algorithms hCSC and saCSC yield significantly improved similarity scores compared to those obtained from giving *Random* datasets as input directly to EMBOSS Needle.
- Algorithms hCSC and saCSC yield similarity scores that are identical or almost identical—notice that the black (Original), green (hCSC), and blue (nCSC/saCSC) points *coincide*—to those obtained from giving *Original* datasets as input directly to EMBOSS Needle.

This implies that algorithms hCSC, nCSC, and saCSC return the rotation maximizing the similarity score for all pairwise comparisons.

Hence, we establish here that the introduced distance measure coupled with the respective algorithms consistently yield a very high accuracy, compared to the standard measure [117, 65, 139], for both *low* and *high* substitution rates.

2.5.2 Time Performance

We then compared the time performance of the algorithms. Each algorithm was given a pair of randomly generated sequences starting from $m = n = 50$ bp and doubling eight times to a length of $m = n = 12,800$ bp. The following order of efficiency was expected, from fastest to slowest:

1. Algorithm saCSC, which runs in $\mathcal{O}(\beta m + n)$ time.
2. Algorithm hCSC, which runs in $\mathcal{O}(\beta(m+n) + \frac{m(m+n)}{\beta})$ time.
3. Algorithm nCSC, which runs in $\mathcal{O}(m(m+n))$ time.
4. Algorithm cNW, which runs in $\mathcal{O}(nm^2)$ time.

Initially, β was set to $\lceil\sqrt{m}\rceil$ and q was set to $\lceil\log_{|\Sigma|} m\rceil$. The results, shown in Figure 2.2, demonstrate orders-of-magnitude superiority of saCSC compared to cNW and nCSC, confirming our theoretical findings. Algorithm hCSC is the second fastest. Although β was set to $\lceil\sqrt{m}\rceil$, saCSC clearly outperforms hCSC, due to the use of a highly optimised implementation of the suffix array construction [63]. This highlights the importance of suitably implemented data structures, such as suffix arrays.

Since the time complexities of hCSC and saCSC depend on β, we repeated the same experiment with these two algorithms, setting β to $\lceil m/25\rceil$ and q to $\lceil\log_{|\Sigma|} m\rceil$—notice that q does not affect the time complexity of the algorithms. The results in Figure 2.2 show that hCSC and saCSC are still the fastest, even though $m = \mathcal{O}(\beta)$, and that saCSC is clearly the fastest of all. As expected for $m = \mathcal{O}(\beta)$, we observe that hCSC and saCSC become gradually slower as m grows. Importantly, notice that varying the value of β relative to m does not significantly affect the time performance of the algorithms in practice.

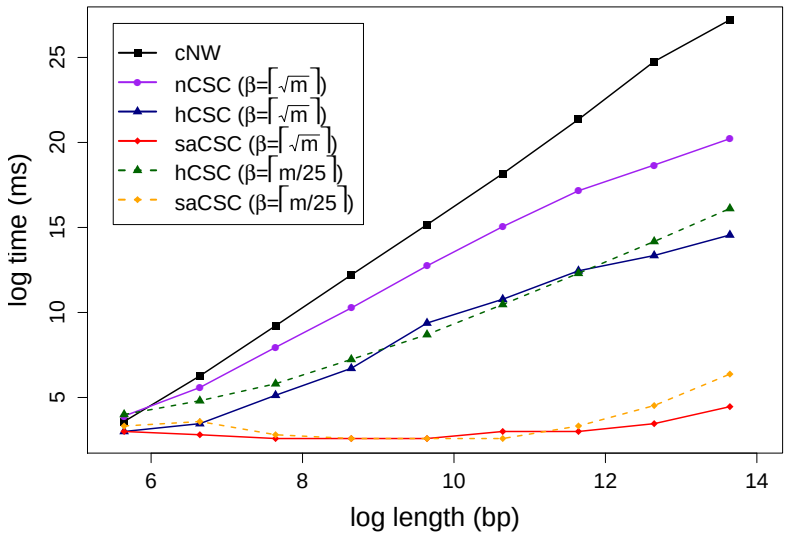


Figure 2.2 Elapsed-time comparison

More algorithms could have been included in the comparison but their (at least) quadratic time complexity [25, 109] prevents them from competing with saCSC.

2.5.3 Application to Synthetic Data

For evaluating the proposed methods for circular sequence comparison in some relevant application, we also implemented the following pipeline for distance-based evolutionary reconstruction of a dataset with N circular sequences:

1. For each pair (X,Y) of the N sequences, we use one method for circular sequence comparison to compute the best rotation X^i.

2. A *similarity* score for (X^i,Y) is then computed using EMBOSS Needle and stored in cell $[X,Y]$ of an $N \times N$ similarity score matrix.

3. The similarity score matrix is transformed into a distance matrix by converting each score into a *distance* relative to the maximum score in the similarity score matrix.

4. Neighbour joining clustering is performed on the distance matrix, using NINJA [180], to produce a phylogenetic tree illustrating predicted evolutionary relationships between all N sequences.

For comparison purposes, phylogenetic trees were also constructed by NINJA [180] for the *Random* datasets using output from the following algorithms:

- cNW (using EMBOSS Needle);
- hSW (see introduction of Section 2.4) followed by EMBOSS Needle to obtain a similarity score;
- saCSCr, with $\beta = 50$, $q = 5$, and $p = 1$, followed by EMBOSS Needle to obtain a similarity score.

Notice that output from cNW should be the same as from EMBOSS Needle with the *Original* datasets as input.

To measure accuracy, the Robinson-Foulds (RF) distance [143, 163] was used to compare the three resultant phylogenetic trees with the tree resulting from EMBOSS Needle on the *Original* and *Random* datasets, denoted by NW(o) and NW(r), respectively. RF distance is defined as the symmetric difference of the clusters in a pair of trees. It is computed iteratively, where removal of an edge in a tree causes the tree to be partitioned in to two clusters. Thus, if two isomorphic trees share the same labelling then they have an RF distance of 0. The results displayed in Table 2.3 clearly show that saCSCr and cNW produce the most accurate results with these nine datasets. As also shown in [112], hSW followed by EMBOSS Needle can often result in sub-optimal global alignments.

To measure time performance, the elapsed time required for each method to process each dataset was recorded and the results are displayed in Table 2.4. It is clear, from the results

Dataset $< \alpha, \gamma, \theta, \kappa, \omega >$	NW(r)	cNW	hSW	saCSCr
$< 12, 2500, 0.05, 0.06, 0.04 >$	16	0	0	0
$< 12, 2500, 0.20, 0.06, 0.04 >$	12	0	0	0
$< 12, 2500, 0.35, 0.06, 0.04 >$	4	0	0	0
$< 25, 2500, 0.05, 0.06, 0.04 >$	44	0	0	0
$< 25, 2500, 0.20, 0.06, 0.04 >$	24	0	0	0
$< 25, 2500, 0.35, 0.06, 0.04 >$	16	0	0	0
$< 50, 2500, 0.05, 0.06, 0.04 >$	86	0	6	0
$< 50, 2500, 0.20, 0.06, 0.04 >$	84	0	0	0
$< 50, 2500, 0.35, 0.06, 0.04 >$	56	0	0	0

Table 2.3 RF distances between the tree obtained from the NW(o) and those obtained from NW(r), cNW, hSW, and saCSCr. The number of sequences in the dataset is denoted by α; γ denotes their lengths; θ denotes the substitution rate; κ and ω denote the relative insertion and deletion rates, respectively.

Dataset $< \alpha, \gamma, \theta, \kappa, \omega >$	cNW	hSW	saCSCr
$< 12, 2500, 0.05, 0.06, 0.04 >$	10139.36	72.43	6.90
$< 12, 2500, 0.20, 0.06, 0.04 >$	9888.84	80.91	6.57
$< 12, 2500, 0.35, 0.06, 0.04 >$	10052.33	80.16	6.28
$< 25, 2500, 0.05, 0.06, 0.04 >$	46311.85	369.02	27.61
$< 25, 2500, 0.20, 0.06, 0.04 >$	46230.07	375.41	28.92
$< 25, 2500, 0.35, 0.06, 0.04 >$	46289.99	400.30	30.44
$< 50, 2500, 0.05, 0.06, 0.04 >$	122165.95	1563.96	125.63
$< 50, 2500, 0.20, 0.06, 0.04 >$	121810.69	1617.89	123.12
$< 50, 2500, 0.35, 0.06, 0.04 >$	120679.32	1662.82	123.77

Table 2.4 Elapsed-time comparison (in seconds) for algorithms cNW, hSW, and saCSCr. The number of sequences in the dataset is denoted by α; γ denotes their lengths; θ denotes the substitution rate; κ and ω denote the relative insertion and deletion rates, respectively.

presented heretofore, that saCSCr outperforms all other algorithms by at least one order of magnitude.

2.5.4 Application to Real Data

We have concluded thus far that using $\beta = \lceil \sqrt{m} \rceil$ and $q = \lceil \log_{|\Sigma|} m \rceil$ results in a reasonable trade-off between running time and accuracy. In the following section, where necessary, we adopt these values and multiply or divide them by a constant factor (of two), depending on the length of the input sequences.

q	β	p	Rotation
5	50	0	566
5	50	1	**578**
5	$\sqrt{m}$	0	567
5	$\sqrt{m}$	1	**578**
5	$2\sqrt{m}$	0	583
5	$2\sqrt{m}$	1	**578**
5	$\frac{\sqrt{m}}{2}$	0	566
5	$\frac{\sqrt{m}}{2}$	1	**578**

Table 2.5 Rotations of GenBank sequence NC_001807 obtained when compared to NC_001643 with varying parameters of saCSCr.

DNA sequences

Pairwise sequence comparison. As the input dataset, we used two real sequences from GenBank [18], a database of nucleotide sequences:

- Human MtDNA sequence of length 16,571 bp (NC_001807).
- Chimpanzee MtDNA sequence of length 16,554 bp (NC_001643).

Their pairwise sequence alignment using EMBOSS Needle gives a similarity of 85.1%. We used cNW, followed by EMBOSS Needle, to obtain the rotation of NC_001807 that maximises its similarity score with NC_001643. This experiment took approximately 28 hours, producing a rotation at position 578 of NC_001807 and improving the similarity score to 91%. This result was then compared to those obtained from saCSC (equivalent to saCSCr with $p = 0$) and saCSCr with varying parameters, displayed in Table 2.5. The convergence of the results after the additional step of refinement (see Table 2.5 in bold) demonstrates the convenience and necessity of saCSCr as compared to saCSC.

For clarity of presentation hereafter, instead of using β, we denote by ℓ the length of the block chosen in algorithm saCSCr.

We repeated this experiment with the human (NC_001807 as before) and gorilla (NC_011120) MtDNA sequences. The MtDNA genome size for NC_011120 is $16,412$ bp. Their pairwise

sequence alignment using EMBOSS Needle gives a similarity of 83.5%. After using saCSCr to rotate sequence NC_001807 ($\ell = 50$, $q = 5$, and $p = 1$), EMBOSS Needle gave a significantly improved similarity of 88.4%.

Finally, note that the experiments which used saCSC and saCSCr each took a fraction of a second to run.

Distance-based Phylogenetic Reconstruction. Three datasets of 16 primate, 12 mammalian and 19 mixed mammalian and primate MtDNA sequences, of average length 16,500 bp, were obtained from GenBank. We followed the same pipeline as described in Section 2.5.3. The RF distance between the following pairs of trees was 0:

- The trees produced by cNW (using EMBOSS Needle);
- The trees produced by saCSCr ($\ell = \lceil\sqrt{m}\rceil = 129$, $q = 5$, and $p = 1$) followed by EMBOSS Needle.

The tree produced for the 12 mammalian sequences is illustrated in Figure 2.3.

RNA sequences

Eighteen viroid sequences were obtained from RefSeq [137], a curated database of molecular sequences. These viroids infect peppers, citrus fruits and other target hosts. Their lengths vary, ranging from 348 to 371 bp. We followed the same pipeline as described in Section 2.5.3. The RF distance between the following two trees was 0:

- The tree produced by cNW (using EMBOSS Needle);
- The tree produced by saCSCr ($\ell = \lceil\sqrt{m}\rceil = 19$, $q = \lceil\log_{|\Sigma|} m\rceil = 5$, and $p = 1$) followed by EMBOSS Needle.

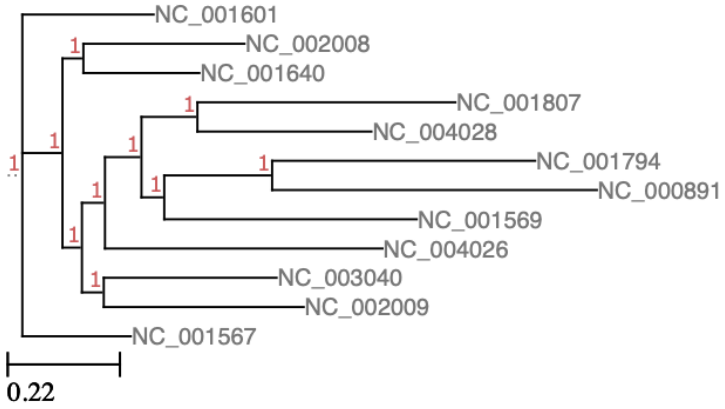


Figure 2.3 Visualisation (using ETE Toolkit [82]) of the phylogenetic tree constructed for 12 mammalian sequences obtained from GenBank and rotated by saCSCr. Each leaf of the tree is labelled with the GenBank accession number of the sequence it represents. Horizontal branch length represents the number of substitutions per base, and thus the evolutionary distance between a leaf node and its ancestor. Labels (in red) at branch splits represent the reliability of a split, where a value of 1 represents 100% reliability.

Protein sequences

Linear, circularly-permuted protein sequences. Eight sequences of proteins, of average length 950 amino acids, belonging to β-glucosidase family [144] were obtained from the UniProt database of protein sequences [39]. We followed the same pipeline as described in Section 2.5.3. The RF distance between the following two trees was 0:

- The tree produced by cNW (using EMBOSS Needle);
- The tree produced by saCSCr ($\ell = \lceil \sqrt{m} \rceil = 31$, $q = \lceil \log_{|\Sigma|} m \rceil = 5$, and $p = 1$) followed by EMBOSS Needle.

Naturally-occurring circular proteins. Ten bacteriocin protein sequences, of average length 20 amino acids, were obtained from Cybase [173], a database of cyclical protein sequences. We followed the same pipeline as described in Section 2.5.3. The RF distance between the following two trees was 0:

- The tree produced by cNW (using EMBOSS Needle);
- The tree produced by saCSCr ($\ell = 2\lceil\sqrt{m}\rceil = 10$, $q = 2\lceil\log_{|\Sigma|} m\rceil = 6$, and $p = 1$) followed by EMBOSS Needle.

2.6 Final Remarks

In this chapter, we introduced a new distance measure for sequence comparison based on q-grams, and showed how it can be applied *effectively* and computed *efficiently* for circular sequence comparison. The most efficient algorithm presented here, saCSC, solves our defined problem CSC, exactly. Extensive experimental results, using both real and synthetic data, show that it maintains an accuracy very competitive to the optimal obtained after considering all rotations of X against Y naïvely using global alignments. We also showed that algorithm saCSCr can bridge the gap between the optimal solution and our approximation via an additional refinement step. Finally, the presented experimental study demonstrates orders-of-magnitude superiority of our approach in terms of runtime efficiency.

Our work [68, 69] has been cited in 15 publications and extended as follows:

- Naturally, our work has been applied to the computation of cyclic edit distance in [9].
- It has been extended to the application of multiple circular sequence alignment in [10].
- In [11], the authors propose filtering techniques to preprocess the sequences given as input to saCSC, which halve the running time of saCSC.

Chapter 3

Motif Discovery

3.1 Introduction

This work has been published in the proceedings of the 25th International Symposium on String Processing and Information Retrieval [85]. The author's contribution to this work was in the formulation of the problem and the design of the algorithm.

3.1.1 Biological Motivation

Motifs are ubiquitous in molecular biological research. A *motif* is a sub-sequence which occurs frequently enough in an alignment of similar or related sequences to be considered conserved. They are, therefore, considered biologically significant and assumed to be associated with a biochemical function. Motifs can be completely conserved and contain no uncertainty, as in Example 14.

Example 14. *The motif inferred trivially from the multiple sequence alignment (MSA, defined in Section 1.3.2) below is* `TGCGTG`*. It represents the DNA sequence with which the Aryl*

hydrocarbon receptor protein (P30561) binds, which is a transcription factor found in mice [39]. Transcription factors are DNA-binding proteins that are involved in the regulation of gene expression [52].

GATAATGCGTGAT

ATCTATGCGTGTC

TGCTCTGCGTGTT

CTGTATGCGTGGT

CTAGGTGCGTGAC

Conversely, more ambiguous motifs contain some positions that represent sets of letters, instead of a single letter, which are known as *degenerate* positions. This type of motif is much more common and defines variation; see Example 15.

Example 15. *The motif inferred from the MSA below is* $CC\{A,T\}^6GG$*, where curly braces represent a set of letters at a degenerate position, and the exponent represents the number of times such a degenerate position is repeated. This motif represents the CArG box DNA sequence with which the Agamous-like MADS-box protein (P29383) binds, which is a predicted transcription factor found in Mouse-ear cress [39].*

AGCTCCAATTAAGGTGACTAG

GCAACCATATTTGGGACAAGC

CTTGCCTTAATAGGCCTTGCT

AAGCCCATTTTTGGAGCCTGT

CTTGCCTATAAAGGCCTTGCT

Most fundamentally, this variation is due to genetic entropy. What is interesting is when such variation causes disease [40].

The Genetic Code

Due to the degeneracy of the genetic code [42], two dissimilar DNA sequences can be translated into two identical protein sequences. Without taking this degeneracy into account, many associations between biological entities can be overlooked. Example 16 highlights the significance of solving problems relating to degeneracy in sequences.

Definition 2. *A codon is a sequence of three nucleotides that codes for one amino acid. See Appendix 2 for a full list of codons and the amino acids to which they are translated.*

Example 16. *The following six DNA codons are all translated into the amino acid Leucine:* `TTA`, `TTG`, `CTT`, `CTC`, `CTA` *and* `CTG` *[42].*

Definition 3. *In genetics, a synonymous substitution mutation in DNA does not cause an alteration in the resulting protein sequence [35].*

Example 17. *If some exon contained the codon* `CTC`*, any substitution in the third position would be synonymous, as the resulting amino acid would always be Leucine.*

Interestingly, codon usage bias has been observed in different species, as illustrated in the Codon Usage Database [116]. Specific notation representing nucleotide ambiguity have long been established by the IUPAC-IUBMB Biochemical Nomenclature Committee [86]. Table 3.1 shows these special symbols, known as *degenerate symbols*, that each represent a non-empty subset of the DNA alphabet, $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Example 18. *The DNA codons from Example 16 can be re-written as* `TTR` *and* `CT`$\diamond$.

Degenerate symbol	Subset of Σ represented by degenerate symbol
Y	{C, T}
R	{A, G}
W	{A, T}
S	{G, C}
K	{T, G}
M	{C, A}
D	{A, G, T}
V	{A, C, G}
H	{A, C, T}
B	{C, G, T}
X, N or ◇	Σ

Table 3.1 Degenerate DNA symbols and the subsets of Σ that they represent.

There are many classes of functional motifs that occur in genomes and examples are discussed below.

DNA Motifs

Transcription factors. The JASPAR open-source database [147] contains transcription factor DNA binding site sequences. The TRANSFAC® database [181] is manually curated and contains such sequences for eukaryotes only.

Example 19. *Activator Protein 1 binds to the following DNA motif:* `TGASTCA` *[3].*

Restriction enzymes. Restriction enzymes recognise specific sequences in DNA at which they cut either both strands or only one strand of the molecule. A comprehensive survey of the different classes of restriction enzymes can be found in [142].

Example 20. *The enzyme Nb.Bpu10I recognises the sequence* $5'$-`CCTNAGC` *and cuts only this strand between the* `C` *and the* `T` *[162].*

Site-specific recombinases. Site-specific recombinases catalyse recombination of DNA molecules by recognising specific motifs. Interestingly, flippase has been shown to bind to sites which vary slightly from its usual recognition sequence, resulting in a reduction in its efficacy [154]. This highlights the importance of considering sequences which vary from the consensus, yet are still biologically meaningful.

Protein Motifs

Motifs are also found in protein sequences and are of great importance when studying protein functions and classifications. PROSITE [157] is a database which stores protein motifs: detailing their sequences, listing the proteins in which they occur, and describing their functions (if known).

Example 21. *The following is an example of a motif taken from PROSITE (PDOC00930 [157]):* `FED{L,V}IA{D,E}{P,A}`*. This motif is found in caveolins; a family of membrane proteins [151, 165]. Currently, this family consists of 80 known proteins, 74 of which contain this motif.*

Motif Discovery

As next-generation sequencing technology advances, there is an increase in the production of genomic data that requires *de novo* assembly and analyses. One such analysis is *motif discovery*, a method of motif prediction [31, 122, 126, 132, 134, 158].

We highlight that the maximal motif discovery problem discussed hereafter differs significantly from the well-established *(ℓ, d)-motif search* problem: find all ℓ-length motifs that occur in at least k sequences from a given collection of sequences, where each occurrence of the motif can contain up to d mismatches [175].

The obvious caveat of (ℓ, d)-motif search approaches [53, 125, 127, 140] is that the length of the motif is restricted and in reality, a longer or shorter motif could be more significant; see Example 22.

Example 22. *Given the sequence* `ACGTTATGTT` *and* $d = 1$*, one should conclude that the significant motif is* `A`$\diamond$`GTT` *rather than, for instance,* `GTT`*; both of which have exactly the same number of occurrences. However, if* $\ell = 3$*, this important observation would be missed.*

We, therefore, focus on the more general problem of *maximal motif discovery*; and also refer the reader to [101], which provides a comprehensive description of the evolution of the maximal motif discovery problem.

Maximal Motif Discovery

A maximal motif $\tilde{M}_{d,k}$ is not determined by a given length, rather, its significance is based on its number of occurrences compared to its factors. A maximal motif is maximal because it cannot be extended to the left or right without reducing its number of occurrences. As importance is given to the number of occurrences, the parameter k sets a minimum threshold for the number of occurrences of a reported maximal motif. The parameter d is more restrictive in that mismatches occur in up to d specific positions in the maximal motif, known as *don't care* symbols and denoted by $\diamond$ (recall Table 3.1). Thus, a maximal motif is also maximal because its don't care symbols cannot be specialised without reducing its number of occurrences.

Importantly, we take a purely *de novo* approach, where only one sequence is needed as input.

We begin with a description of recent results in maximal motif discovery. We follow this with an explanation and formal definition of the specific problem considered in this chapter.

3.1.2 Previous Work

We highlight the fact that the motif discovery problem is one of the oldest problems in bioinformatics, and there is exist over a hundred algorithms to solve this problem. Therefore, this section is by no means exhaustive, and focuses on recent and/or popular solutions. A survey of some probabilistic and combinatorial solutions is provided in [148].

Probabilistic Solutions

Many commonly used motif discovery tools exist which use probabilistic methods, such as those presented in [14, 24, 29, 30, 61, 76, 77, 88, 90, 93, 98, 136, 141, 153]. We refer the reader to [49, 80, 103, 168] which discuss and evaluate such tools.

Combinatorial Solutions

A thorough overview of the first generation of combinatorial solutions for the (maximal) motif discovery problem is provided in [22]. There also exist algorithms to discover variations and extensions of maximal motifs, including motifs with (flexible [5, 46]) gaps [6, 45, 53, 84, 119]; as well as algorithms to discover *bases*, sets of strings that represent all (maximal) motifs [119, 120, 128, 129].

In [87], the authors present a three-stage algorithm to report all maximal motifs given a set S of sequences, summarised in the following:

Comparison. A similarity matrix is built following the all-against-all pairwise comparison of all overlapping factors of length L of all sequences in S. The metric used to compute similarity is chosen by the user.

Clustering. Similar factors of length L are clustered to form *elementary motifs*.

Convolution. The occurrences $\mathsf{occ}(f)$ of factors f of length ℓ of all elementary motifs, for all $\ell \in [1, L+1]$, are compared, and redundant factors are eliminated resulting in a set of maximal factors. Based on $\mathsf{occ}(f)$, these factors are then combined to form maximal motifs.

In [101], the authors present an algorithm to report all motifs of length $L \geq m$ given that there are at least k occurences of (ℓ, d)-variants of the motif in the set of input sequences, where an (ℓ, d)-variant of a motif is a string of length ℓ with Hamming distance d from a factor of length ℓ of the motif; and $\ell \in [m, L)$. The algorithm runs in $\mathscr{O}(tn\Sigma^L)$ time and $\mathscr{O}(\Sigma^L + tn)$ space, where t is the number of input sequences, n is the length of each sequence, and L is the maximum length of a reported motif.

A class of solutions for the maximal motif discovery problem make use of suffix trees [32–34, 53, 127, 146, 171, 172], and many are output-sensitive [7, 72, 121]. The algorithm proposed in [70, 71] makes use of both of these and is, to the best of our knowledge, the most recent combinatorial solution for maximal motif discovery. The authors present a data structure termed a *motif trie*, which represents all prefixes, suffixes, and occurrence positions of each maximal motif $\tilde{M}_{d,k}$ in the set $\mathscr{M}_{d,k}$ of maximal motifs. Specifically, each edge represents a deterministic, maximal (possibly empty) factor of the input string (obtained from a suffix tree), and each node implies a *don't care* symbol. Each branch of the motif trie represents a maximal motif, where a suffix of a maximal motif that is also maximal itself is represented as a separate branch in the trie. The authors present an algorithm with an

output-sensitive overall time complexity of

$$\mathscr{O}(nd + d^3 \cdot \sum_{\tilde{M}_{d,k} \in \mathscr{M}_{d,k}} |\mathsf{occ}(\tilde{M}_{d,k})|)$$

assuming the input sequence of length n is built on a constant-sized alphabet.

Problem Definition and Contribution

Motivated by the aim of discovering interesting regions in large genomic sequences, in this chapter, we propose a data structure as sensitive as the motif trie, and crucially, that has the additional advantage of being a dynamic data structure: the *motif graph*. We also emphasise the space-efficiency of the motif graph in comparison to the motif trie, where the former does not store redundant data for maximal suffixes of maximal motifs. The motivation behind creating a dynamic structure was to facilitate a sliding window on the input sequence. Specifically, this ensures the additional ability to find interesting ℓ-length regions of the sequence, which is useful in various bioinformatics applications, including the prediction of the origin of chromosomal replication (OriC) [111].

Example 23. *The length of OriC in model bacterial species ranges from 120 to 300bp; for example, it is 240bp in E. coli [100]. Furthermore, motifs that occur within OriC, such as DnaA boxes, show that d and k are small constants in practice; for example, $d = 2$ and $k = 4$ [62].*

Before presenting the problem formally, let us denote by $\mathscr{M}_{i,d,k}$ the set of the maximal motifs in the ℓ-length window ending at position i in string X, each of which must occur at least k times in the window and contain at most d don't care symbols.

MAXIMAL MOTIF DISCOVERY IN A SLIDING WINDOW (MMDSW)

Input: A string X of length n and integers ℓ, $k > 1$ and d.

Output: An array $\mathscr{S}_X$ of scores, where $\mathscr{S}_X[i] = |\mathscr{M}_{i,d,k}|$ and $i \in [\ell, n)$.

We present the first on-line algorithm to find the occurrences of all maximal motifs in a sliding window in

$$\mathscr{O}(nd\ell + d\lceil\frac{\ell}{w}\rceil \cdot \sum_{i=\ell}^{n-1} |\Delta_{i-1}^{i}|)$$

time, where Δ_{i-1}^{i} is the symmetric difference of the sets of occurrences of maximal motifs in $X[i-\ell\,..\,i-1]$ and in $X[i-\ell+1\,..\,i]$, and w is the size of the machine word. The machine word is the unit of data handled by computer processors. It is of constant size, ranging from 32 to 64 bits for modern processors. The space complexity of our algorithm is $\mathscr{O}(\ell^2)$. This result poses an improvement over the time required to solve Problem MMDSW using the motif trie [71], which would then be

$$\mathscr{O}(nd\ell + d^3 \cdot \sum_{i=\ell, \tilde{M}_{d,k} \in \mathscr{M}_{i,d,k}}^{n-1} |\mathsf{occ}(\tilde{M}_{d,k})|).$$

This improvement is significant as a single occurrence of a maximal motif would be reported $\mathscr{O}(\ell)$ times by the latter approach. Therefore, the proposed algorithm results in a speed-up of $\mathscr{O}(d^2 w)$ per occurrence of a maximal motif. Finally, note that our algorithm can be modified trivially to report the actual occurrences of all maximal motifs instead of array $\mathscr{S}_x$.

3.1.3 Chapter Summary

In summary, our contribution in this chapter is twofold.

Data Structure. In Section 3.2, we define the motif graph which can be used to find the set $\mathscr{M}_{d,k}$ of maximal motifs in an input string, given parameters k and d.

Algorithm. In Sections 3.3, we describe the effect of sliding the window on $\mathscr{M}_{d,k}$, the motif graph and the score of the window. An analysis of the presented algorithm is provided in Section 3.5.

3.2 Definitions

Maximal Motifs

Recall the definition of a suffix tree from Section 1.3.2. Each factor of a string X is uniquely represented by the path-label of a node of the suffix tree ST_X of X. More specifically, each *right-maximal* repeated factor of X is uniquely represented by an internal node of ST_X. In other words, at least one occurrence of this factor (of which there are at least two) in X is followed by a letter that is distinct from the rest. Therefore, a right-maximal factor of X cannot be extended to the right without reducing its number of occurrences. See Example 24 for examples.

A right-maximal factor is also *left-maximal* if the preceding letter of at least one of its occurrences in X is distinct from the rest. If the number of occurrences of a suffix target is not equal to that of its suffix origin, or if it is a target of multiple suffix links, then the suffix target is left-maximal. That is, it cannot be extended to the left without reducing its number of occurrences. If a node is left-maximal, all of its ancestors are also left-maximal.

Henceforth, we will refer to repeated factors that are both left- and right-maximal as *seeds*, and their corresponding suffix tree nodes as *seed nodes*. Refer to Example 24 for examples of seed nodes.

Remark 3.2.1. *Every seed node is an internal node in a suffix tree, but not every internal node necessarily corresponds to a seed. The number of seeds are thus $\mathcal{O}(n)$ because the number of nodes in a suffix tree are $\mathcal{O}(n)$.*

Example 24 (Running example)**.** *The suffix tree in Figure 3.1 is built from the string*

$$X = \texttt{AGCTAGTTCTAGCTAGCTAG}$$

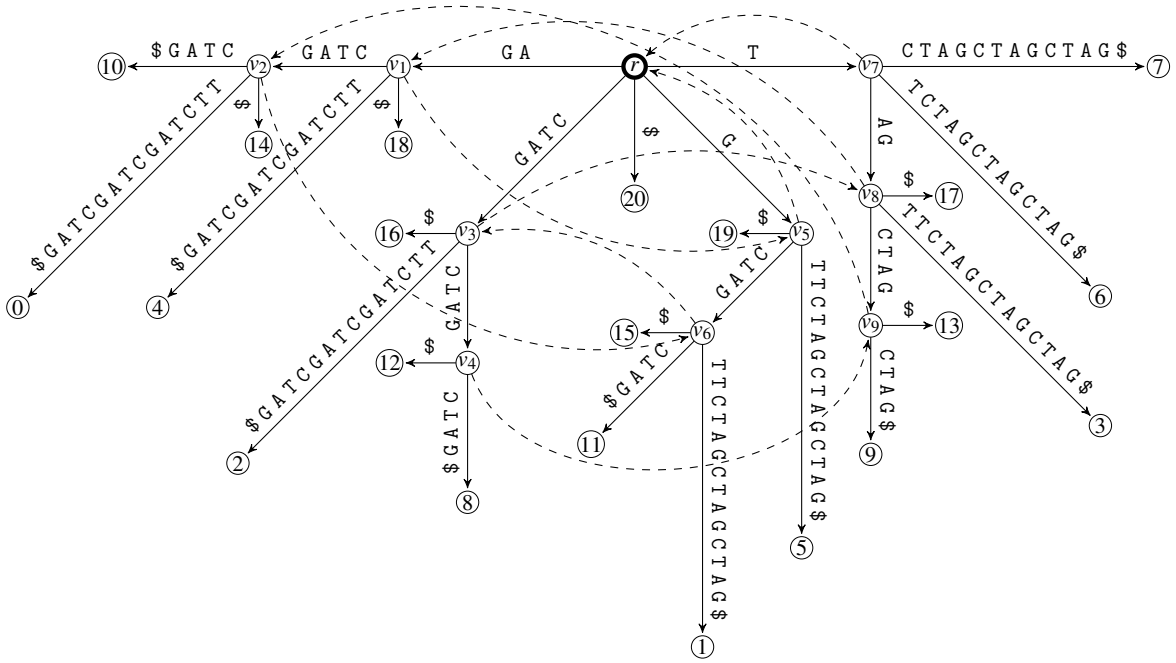
built on Σ and concatenated with $\$ \notin \Sigma$. In order to deduce which nodes are seed nodes, we begin with a list of candidate seed nodes comprised of the set of all internal nodes, $\{V_1,..,V_9\}$, which represents the right-maximal repeated factors of X. For example, V_3 represents $\texttt{CTAG}$ and V_5 represents $\texttt{G}$. For all candidate nodes that are suffix targets, we check their left-maximality as follows. For each pair $\langle s(V_i),V_i\rangle$ of suffix targets $s(V_i)$ and suffix origins V_i, we check if their number of occurrences match. Note that we proceed with this step in ascending order of factor length $D(s(V_i))$, as follows:

$$\{\langle V_5,V_1\rangle,\langle V_1,V_8\rangle,\langle V_8,V_3\rangle,\langle V_3,V_6\rangle,\langle V_6,V_2\rangle,\langle V_2,V_9\rangle,\langle V_9,V_4\rangle\}$$

As $|occ(V_5)| = 5 = |occ(V_1)|$, V_5 is not a seed and is deleted from the list. However, as $|occ(V_1)| = 5 \neq |occ(V_8)| = 4$, V_1 is a seed and remains in the list; and so on. What remain in the list are all seed nodes: $\{V_1,V_2,V_3,V_4,V_7\}$.

For our purposes and given thresholds d and k, we define a *motif* $\tilde{M}_{d,k}$, that occurs in a given string X, as a sequence of d' don't care symbols and up to $d'+1$ seeds, where $d' \in [0,d]$ and $|\mathsf{occ}(\tilde{M}_{d,k})| \geq k$. A motif $\tilde{M}_{d,k}$ is *maximal* as it cannot be extended (to the left or right) nor can it be specialised (that is, its don't care symbols cannot be replaced with symbols from Σ) without reducing its number of occurrences [70, 71, 87]. Each motif must begin and

Figure 3.1 The suffix tree of the string $X =$ AGCTAGTTCTAGCTAGCTAG concatenated with $\$ \notin \Sigma$. Each leaf node has been labelled with the index i of the suffix $X[i..n-1]$ that it represents, where $i \in [0,n)$. All other explicit nodes are labelled V and root node r is outlined in bold. Suffix links are shown as dashed directed edges.



end with a seed, as $\mathsf{occ}(\tilde{M}_{d,k})$ is not affected by flanking $\tilde{M}_{d,k}$ with don't care symbols [71]. Importantly, a singleton seed can be a motif, where $d' = 0$; this will be referred to as a *singleton motif*. We denote the set of all maximal motifs in X as $\mathscr{M}_{d,k}$; see Example 25 for examples.

Motif Graph

It is evident that the suffix tree is useful in finding seeds and we now explain how it can be augmented to find motifs. We term this augmented suffix tree a *motif graph*. Each internal node V of ST_X is decorated with the following:

- an integer variable $|\mathsf{occ}(V)|$, which holds the number of occurrences of V in X;

- a Boolean variable $\mathsf{isSeed}(V)$, which is TRUE if node V is a seed, and FALSE otherwise.

Each node V where $\mathsf{isSeed}(V) = \text{TRUE}$ is further augmented with the following:

- a Boolean variable $\mathsf{isMotif}(V)$, which is TRUE if node v represents a singleton motif, and FALSE otherwise;
- a bit-vector $\mathscr{B}(V)$, of total size n bits, which indicates the occurrence positions of V in X.

A labelled, directed edge $U \xrightarrow{d'} V$, from seed node U to seed node V, indicates that $U \diamond^{d'} V$ occurs at least k times in string X, where $d' \in (0,d]$. Note that $U = V$ is possible and would appear as a loop in the motif graph. We will refer to extra edges as *motif edges*, and reiterate their distinction from suffix tree edges. In order to facilitate the construction of motif edges, we store an array E, of linear size, in which each element keeps a list of pointers to seed nodes which have an occurrence ending at the corresponding position in X.

Definition 4. *A bit-wise shift operation shifts a bit-vector by a defined number of bits, in either the right or left direction. A bit-wise and operation compares corresponding bits in a pair of bit-vectors of the same length. If both bits in a corresponding pair of bits are set (1), the corresponding bit in the output bit-vector will also be set; otherwise it will be clear (0).*

We define the bit-vector $\mathscr{B}(U \xrightarrow{d'} V)$ of a motif edge as the bit-vector resulting from a shift-and operation of $\mathscr{B}(U)$ and $\mathscr{B}(V)$, which represents the starting positions of occurrences of $U \diamond^{d'} V$ in X. Note that the shift accounts for both d' and $D(U)$, where $D(U)$ is the depth of the node U and thus the length of the factor that it represents. We now define the symbol $\equiv$ for bit-vectors as follows. For a given d', if every occurrence of U is succeeded by $\diamond^{d'} V$, and every occurrence of V is preceded by $U \diamond^{d'}$, then $\mathscr{B}(U) \equiv \mathscr{B}(V) \equiv \mathscr{B}(U \xrightarrow{d'} V)$. We denote

as $\mathsf{indegree}(U)$ the number of incoming motif edges to the seed node U. Similarly, we denote as $\mathsf{outdegree}(U)$ the number of outgoing motif edges from the seed node U.

Each motif corresponds to a *maximal path* of motif edges, where the maximality is satisfied by the following two conditions with respect to d and k. Suppose

$$\tilde{M}_{d,k} = U \diamond^{d'_1} Z_1 \diamond^{d'_2} Z_2 \,..\, \diamond^{d'_{q-1}} Z_{\text{q-1}} \diamond^{d'_q} Z_{\text{q}}$$

corresponds to a path

$$P = U \xrightarrow{d'_1} Z_1 \xrightarrow{d'_2} Z_2 \,..\, \xrightarrow{d'_{q-1}} Z_{\text{q-1}} \xrightarrow{d'_q} Z_{\text{q}}$$

then $\sum_{i=1}^{q} d'_i \leq d$ and $|\mathsf{occ}(P)| = |\mathsf{occ}(\tilde{M}_{d,k})| \geq k$. In other words, P is maximal if the resultant motif occurs at least k times in the string and contains no more than d don't care symbols.

The longest prefix of a path P, denoted by $\mathsf{prefix}(P)$, is the path resulting from the truncation of the last edge in P. If $|\mathsf{occ}(P)| \neq |\mathsf{occ}(\mathsf{prefix}(P))|$, then $\mathsf{prefix}(P)$ is also a maximal path and hence corresponds to a motif. Similarly, the longest suffix of P, denoted by $\mathsf{suffix}(P)$, is the path resulting from the truncation of the first edge in P. If $|\mathsf{occ}(P)| \neq |\mathsf{occ}(\mathsf{suffix}(P))|$, then $\mathsf{suffix}(P)$ is also a maximal path and hence corresponds to a motif.

The bit-vector $\mathscr{B}(P)$ of a path P can be computed by a series of bit-wise shift-and operations of the bit-vectors of the edges in P.

Example 25 (Running example)**.** *From Example 24, we have the following set of seeds:*

$$\{V_1 = \mathtt{AG}, V_2 = \mathtt{AGCTAG}, V_3 = \mathtt{CTAG}, V_4 = \mathtt{CTAGCTAG}, V_7 = \mathtt{T}\}$$

For $d = 1$ and $k = 2$, we find the following set $\mathscr{M}_{1,2}$ of motifs from the motif graph shown in Figure 3.2.

$\tilde{M}_{1,2}$	AG	$AG\diamond T$	$AGCTAG$	$AGCTAG\diamond T$	$CTAG$	$CTAG\diamond T$	$CTAGCTAG$
$\lvert occ(\tilde{M}_{1,2})\rvert$	5	4	3	2	4	3	2

3.3 Sliding Window

The main computational challenge in reporting motifs in a sliding window is maintaining the left- and rightmost seeds in two respects: checking their maximality (nodes) and updating their relationship with neighbouring seeds (edges). These changes to the motif graph identify and thus efficiently update *only* the subset of motifs occurring at both ends of the window.

Specifically, in what follows, we describe the effect on $\mathscr{M}_{i,d,k}$ and the motif graph when adding a letter to the right of the window, and deleting a letter from the left, thus simulating the sliding window on X.

3.3.1 Update of Set of Motifs for Sliding Window

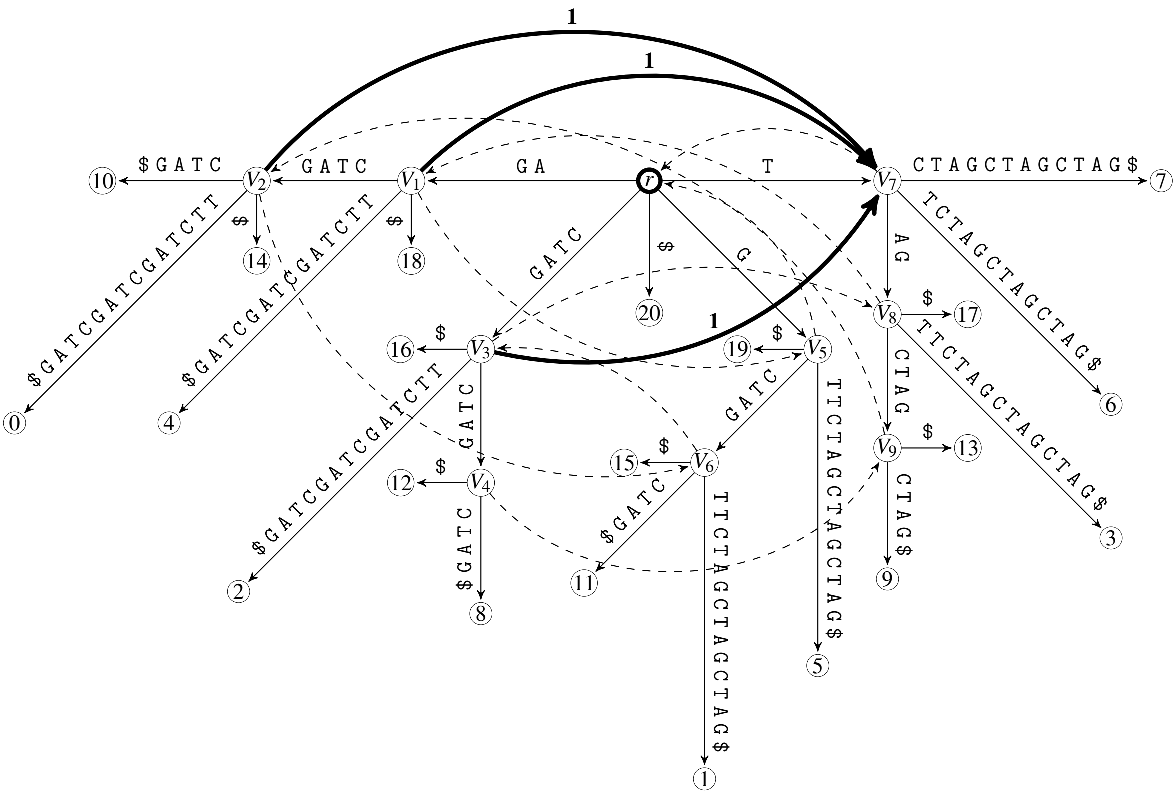
Before proceeding with updates, the set $\mathscr{M}_{i,d,k}$ is copied from the set $\mathscr{M}_{i-1,d,k}$ from the previous window.

Adding a Letter to the Right

When adding a letter $X[i] = \alpha$ to the right of the window, the following cases are checked in order, if and only if $|\mathsf{occ}(\alpha)| \geq k$ in the window.

1. If α now extends at least k occurrences of some motif $\tilde{M} \in \mathscr{M}_{i,d,k}$, it becomes the suffix of a new motif $\tilde{M}' = \tilde{M} \diamond^{d'} \alpha$, which occurs at least k times in the window, where $d' \in [0,d]$. In this case, the new motif $\tilde{M}'$ is added to $\mathscr{M}_{i,d,k}$. If the number of occurrences of $\tilde{M}$ is equal to $\tilde{M}'$, $\tilde{M}$ is deleted from $\mathscr{M}_{i,d,k}$, as it is no longer maximal.

Figure 3.2 The motif graph of the string $X =$ `AGCTAGTTCTAGCTAGCTAG` concatenated with $\$ \notin \Sigma$. Each leaf node has been labelled with the index i of the suffix $X[i\,..\,n-1]$ that it represents, where $i \in [0, n)$. All other explicit nodes are labelled V and root node r is outlined in bold. Suffix links are shown as dashed directed edges. Motif edges and their labels are bold.



$\mathscr{B}_{V_1}$	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
$\mathscr{B}_{V_2}$	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
$\mathscr{B}_{V_3}$	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0
$\mathscr{B}_{V_4}$	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
$\mathscr{B}_{V_7}$	0	0	0	1	0	0	1	1	0	1	0	0	0	1	0	0	0	1	0	0	0

2. The letter α can be added to $\mathscr{M}_{i,d,k}$ as a singleton motif if and only if it is not already in $\mathscr{M}_{i,d,k}$ and one of the following is true:

   - No motifs were added to $\mathscr{M}_{i,d,k}$;
   - One motif $\tilde{M}'$ was added to $\mathscr{M}_{i,d,k}$ and $|\mathsf{occ}(\alpha)| > |\mathsf{occ}(\tilde{M}')|$;
   - Two or more motifs were added to $\mathscr{M}_{i,d,k}$.

Deleting a Letter from the Left

When deleting the leftmost letter α of the window, every motif $\tilde{M}' = \alpha\tilde{M} \in \mathscr{M}_{i,d,k}$ must be deleted if now $|\mathsf{occ}(\tilde{M}')| < k$ in the window. After this possible deletion, the following cases are considered:

- If $\tilde{M} = \varepsilon$, and thus $\tilde{M}' = \alpha$, then nothing more is done.
- If $\tilde{M} \neq \varepsilon$, then $\tilde{M}$ is added to $\mathscr{M}_{i,d,k}$, if and only if $\tilde{M} \notin \mathscr{M}_{i,d,k}$ and it is not a prefix of a motif $\tilde{M}'' \in \mathscr{M}_{i,d,k}$ such that $|\mathsf{occ}(\tilde{M}'')| = |\mathsf{occ}(\tilde{M})|$.

3.3.2 On-line Update of Suffix Tree for a Sliding Window

As the motif graph is fundamentally a suffix tree, we will first provide an overview of how a letter can be added to the right and deleted from the left, namely by Ukkonen's [170] and Senft's [155] algorithms, respectively.

Adding a Letter to the Right

The following is a summary of the relevant details of Ukkonen's algorithm; for further details, refer to [170]. The algorithm iteratively builds ST_X, where the tree ST_{X_i} at each iteration i represents the implicit suffix tree of the prefix $X[0..i]$ of X, where only j suffixes are

represented as leaf nodes and leaf $j-1$ is the most recent leaf added to the suffix tree, where $0 \leq j-1 \leq i < n$. If $j-1 = i$, the suffix tree ST_{X_i} is explicit. The algorithm makes use of a special pointer to a node known as the *active point* $\mathscr{A} = \langle \gamma, \mu, \lambda \rangle$ that corresponds to the longest repeated suffix of $X[0..i]$. Recall from Section 1.3.2 that μ is the number of implicit nodes skipped on the edge from explicit node γ, and the edge label ends with $\alpha = X[\lambda]$.

Example 26 (Running example)**.** *The implicit suffix tree of* $X =$ `AGCTAGTTCTAGCTAGCTAG` *is shown in Figure 3.3. The longest implicit suffix* $X[12..19] =$ `CTAGCTAG` *is the path from the root to the active point,* $\mathscr{A} = \langle V_3, 4, 15 \rangle$*. In other words, the active point points to an implicit node* $\langle V_3, 4, 15 \rangle$ *which exists between explicit nodes* V_3 *and* 8*, and is shown as a black dot in Figure 3.3. This implicit suffix tree can be transformed into an explicit suffix tree, shown in Figure 3.1, by appending* \$ *to* X*.*

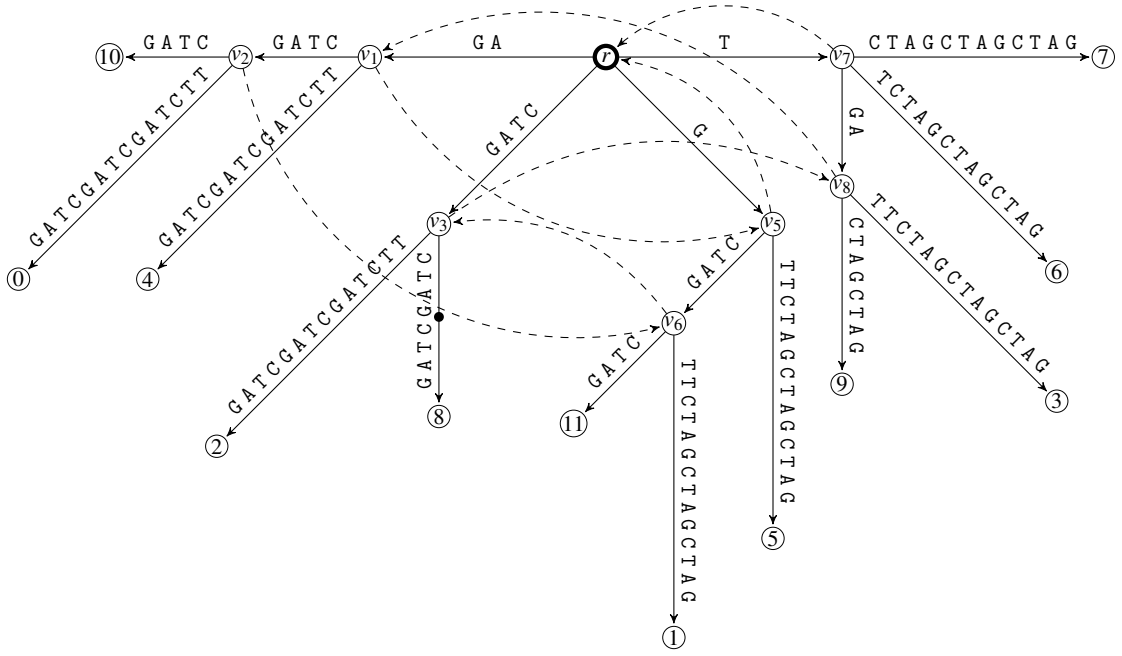
When adding a letter $X[i] = \alpha$ to the right of the string, and thus extending each existing suffix $\beta \in \{\beta_0, \ldots, \beta_{i-1}\}$, the following three rules are used:

- If there is no path from the root representing $\beta\alpha$, we do either of the following. Let $P(V) = \beta$:
  - If V is a leaf node, we simply append α to the label of the edge ending at V. This is known as a Rule 1 extension.
  - If V is not a leaf node, observe that $\mathscr{A}$ is pointing to V. We create a new leaf node U, for which the label of its incoming edge from V is α. Thus, U represents the j^{th} suffix of X. If V is an implicit node, it becomes explicit and the active point $\mathscr{A}$ becomes $\langle V, 0, i \rangle$. This is known as a Rule 2 extension.
- If $\beta\alpha$ already exists within the suffix tree, the structure of the tree is unchanged and μ is incremented by one. However, if $\mathscr{A}$ reaches an explicit node V', then $\mathscr{A} = \langle v', 0, i \rangle$.

This is known as a Rule 3 extension and we say that $\mathscr{A}$ is *moving along* a branch of ST_X, as i moves away from j.

If a Rule 2 extension results in making a node V explicit, then it becomes the suffix origin of a newly created suffix link. Additionally, further leaf nodes may need to be added by moving $\mathscr{A}$ to $s(V)$. In this case, we say that $\mathscr{A}$ is *following a suffix link*, as j moves closer to i.

Figure 3.3 The implicit suffix tree of the string $X =$ `AGCTAGTTCTAGCTAGCTAG`. Each leaf node has been labelled with the index of the suffix j that it represents, for all suffixes $j \in [0, 12)$. All other explicit nodes V are labelled as in Figure 3.1 and root node r is outlined in bold. Note that nodes V_4 and V_9 from Figure 3.1 do not exist here. Suffix links are shown as dashed directed edges. The active point is a black dot on the path between nodes V_3 and node 8.



Example 27 (Running example)**.** *Observe that, for example, suffixes* 10 *and* 11 *were added as leaf nodes when letter* $X[16] =$ `C` *was added.*

Deleting a Letter from the Left

The following is a summary of the relevant details of Senft's algorithm; for further details, refer to [155]. When deleting a letter α from the left of the window, the longest unique prefix of the window is being deleted, which in fact represents the whole window. In doing so, all repeated prefixes of suffixes must be maintained, by giving importance to the longest repeated prefix β, where $\beta[0] = \alpha$. Let $P(V) = \beta$ be the longest repeated prefix and $P(U) = \beta\delta$ be the longest unique prefix; thus V is an ancestor of U. We have two possible cases, as follows:

1. If V is an explicit node, we simply delete the leaf U. If V now only has one remaining child U', we delete V and merge the edge that was directed at V with the edge that is directed at U'.

2. If V is an implicit node, then β is also the longest repeated suffix; thus $\mathscr{A}$ is pointing to V. We delete the leaf U and node V becomes a leaf node, representing suffix j. Consequently, j is incremented by one and $\mathscr{A}$ is moved to its longest proper suffix.

The representation of edge labels can be updated efficiently due to the correctness of their offsets relative to the start of the window. Thus, a batch update of all labels can be done after every ℓ window shifts.

Example 28 (Running example)**.** *Considering Figure 3.3, the deletion of the leftmost letter (*A*) would result in the deletion of leaf node* 0 *which represents the longest suffix. This would cause the subsequent deletion of node V_2 as it would have one remaining child; the edges from V_1 to V_2 and V_2 to leaf node* 10 *would be merged.*

3.4 Algorithm

The proposed algorithm works in an on-line manner. That is, the algorithm processes the input in a serial fashion, such that the entirety of the input is not processed in one step. In order to facilitate this efficiently, we begin by appending a string of ℓ letters not in Σ to the start of X. We also append a unique letter not in Σ to the end of X to ensure that the motif graph (augmented suffix tree ST) of the final window is explicit. We therefore run the algorithm on the new string $X_\$$. As well as X, the algorithm requires the following input parameters:

- The length n of the string X;
- The length ℓ of the window on X;
- The maximum number d of allowed don't care symbols;
- The minimum number k of occurrences of maximal motifs.

First, we must redefine and extend our original definition of the motif graph as follows. Each internal node V of ST is decorated with the following:

- an integer variable $|\mathsf{occ}(V)|$, which holds the number of occurrences of V in the window of length ℓ on $X_\$$.
- a Boolean variable $\mathsf{isSeed}(V)$, which is TRUE if node V is a seed in the window, and FALSE otherwise.

Each node V where $\mathsf{isSeed}(V) = \text{TRUE}$ is further augmented with the following:

- a Boolean variable $\mathsf{isMotif}(V)$, which is TRUE if node V represents a singleton motif in the window, and FALSE otherwise.

- a bit-vector $\mathscr{B}(V)$, of total size ℓ bits, which indicates the occurrence positions of V in the window. In order to maintain $\mathscr{B}(V)$ efficiently, we introduce an integer variable pivot(V), which acts as an anchor so that $\mathscr{B}(V)$ is only updated when an occurrence of V is added or deleted, rather than in every step i of the algorithm.

Additionally, E (defined previously in Section 3.2) is now a dynamic structure, of size $\mathscr{O}(\ell)$, in which each element keeps a list of pointers to seed nodes which have an occurrence ending at the corresponding position relative to the current window. The aforementioned definition of a motif edge remains; however, the shift operation that produces its bit-vector now also takes the pivot into consideration.

Before proceeding, we define functions used throughout Algorithm MMDSW:

- Given a bit-vector $\mathscr{B}$, the function popCount($\mathscr{B}$) returns the number of set bits in $\mathscr{B}$.
- Similarly, the Boolean function kPopCount($k, \mathscr{B}$) returns TRUE if the number of set bits in $\mathscr{B}$ is at least k; and FALSE otherwise.
- The function getEndPositions() returns a list E' to which, for each d' and for each pointer to a node V in the list at position $j' = j - d' - 1$ in E, a pair $\langle V, j' \rangle$ is added, where $d' \in (0, d]$ and $j-1$ is the most recent leaf added to the suffix tree (as described in Section 3.3.2).

In the following, we provide pseudocode for Algorithm MMDSW; where the addition symbol, with respect to strings, defines concatenation. The pseudocode for all functions called by Algorithm MMDSW can be found in Appendix 1.

```
1 Algorithm MMDSW(X, n, ℓ, d, k)
2     X_$ ← $^ℓ + X + $;
3     n_$ ← n + ℓ + 1;
4     initialise empty ST;
5     for i ← 0 to n_$ − 1 do
6         rightHandSide(i);
7     return;
```

Importantly, note that the algorithm assumes that the window is moving with respect to j. Therefore, the addition of leaf j to the motif graph triggers the deletion of leaf $j-\ell$, and the score is updated accordingly. Thus the score of a window represents the number of motifs that occur in the window, such that each motif has at least k starting positions in $X_\$[j-\ell+1\,..\,j]$.

3.4.1 Reading a Letter on the Right

Recall that the active point $\mathscr{A}$ represents the rightmost seed that has an occurrence starting at position j and ending at position i of $X_\$$. The same extra data structures are therefore stored for $\mathscr{A}$ as for internal nodes. In the following, we describe the various cases of how $\mathscr{A}$ is updated when reading letter $X_\$[i]$. Pseudocode relating to the following is provided in Function 3, where on Line 2, the function readLetter(i) reads the letter at position i in the string $X_\$$ and returns the updated value of the active point $\mathscr{A}$ of the suffix tree.

The active point $\mathcal{A}$ starts moving along a new branch from the root r.

The active point $\mathscr{A} = \langle r, 1, i\rangle$ is initialised, where isSeed$(\mathscr{A})$ = TRUE and $|\mathsf{occ}(\mathscr{A})|$ is computed by incrementing the number of occurrences of explicitChild$(\mathscr{A})$ by one (Lines 2 & 3,

Function 4). If $|\mathsf{occ}(\mathscr{A})| \geq k$, then $\mathscr{B}(\mathscr{A})$ and $\mathsf{pivot}(\mathscr{A})$ are copied from $\mathsf{explicitChild}(\mathscr{A})$ and shifted to accommodate the extra occurrence at position j (Lines 4-5, Function 4).

Incoming motif edges are then added to $\mathscr{A}$ in the following way:

1. The list E' is obtained by calling $\mathsf{getEndPositions}()$ (Line 6, Function 4).

2. Each pair in E' corresponds to a potential motif edge $V \xrightarrow{d'} \mathscr{A}$ (Lines 2-3, Function 5). For each such pair in E', if $V \diamond^{d'} \mathscr{A}$ does not exist at least k times in the window, such that $\mathsf{kPopCount}(k, \mathscr{B}(V \xrightarrow{d'} \mathscr{A}))$ is FALSE, then the edge is discarded (Lines 4-5, Function 5).

3. If they exist, the remaining potential motif edges are clustered with respect to $|\mathsf{occ}(V \diamond^{d'} \mathscr{A})|$ (Line 6, Function 5). In each cluster, the edges are sorted with respect to $D(V)$, and the motif edge $V \xrightarrow{d'} \mathscr{A}$ is added to the motif graph from the deepest node V (Lines 7-9, Function 5). This is done in order to ensure left- and right-maximality. For $d' > 1$, a motif edge can only be added if there is no motif edge from a descendant seed node of V to $\mathscr{A}$ with the same bit-vector as $V \xrightarrow{d'} \mathscr{A}$ (Line 10, Function 5). Importantly, this is regardless of the label d' of either edge and ensures that $V \diamond^{d'} \mathscr{A}$ cannot be specialised (that is, its don't care symbols cannot be replaced with symbols from Σ) without reducing $|\mathsf{occ}(V \diamond^{d'} \mathscr{A})|$ (the second condition of maximality).

After the possible addition of motif edges, $\mathsf{isMotif}(\mathscr{A})$ is set to FALSE if there is any edge $V \xrightarrow{d'} \mathscr{A}$ such that $\mathscr{B}(\mathscr{A}) \equiv \mathscr{B}(V \xrightarrow{d'} \mathscr{A})$, where $d' \in (0, d]$. An update is made to $\mathsf{isMotif}(V)$ in a similar way. This step corresponds to Function 8 which is called in Line 13 in Function 4.

The active point $\mathcal{A}$ advances along a branch and reaches an internal node U.

The active point $\mathscr{A} = \langle U, 0, i \rangle$ points to an internal node U, which is updated as follows (Lines 6-7, Function 3). First, $|\mathsf{occ}(U)|$ is incremented by one and $\mathsf{isSeed}(U)$ is updated (Lines 2-3, Function 9). If $\mathsf{isSeed}(U) = \text{TRUE}$ and $|\mathsf{occ}(U)| \geq k$, then seed node U is updated by copying missing information from $\mathscr{A}$ regarding its new occurrence at position j, before the extra structures and incoming edges of $\mathscr{A}$ can be deleted (Line 4, Function 9). Specifically, $\mathscr{B}(U)$ and $\mathsf{pivot}(U)$ are copied from $\mathscr{A}$ (Lines 5-6, Function 9). In order to record the ending position of the occurrence of U starting at position j, a pointer to U is added to the list of pointers representing position $j + D(U) - 1$ in E (Line 7, Function 9).

Then, for each incoming motif edge to $\mathscr{A}$, the motif edge is copied to U if it does not already exist. If the motif edge already exists, it is not duplicated, however, it may affect whether U is a singleton motif. This corresponds to Lines 8-11 in Function 9 which call Functions 10 and 8.

The active point $\mathcal{A}$ moves further along a branch and passes by a seed node U.

The active point, which is now $\mathscr{A} = \langle U, 1, i \rangle$, is updated in the same way that is described when $\mathscr{A} = \langle r, 1, i \rangle$ (Lines 2-5, Function 4).

Incoming motif edges do not need to be computed *ab initio* as they are copied or taken from node U (Function 11). However, E' must still be computed in order to consider only those edges to U that can also exist to $\mathscr{A}$, as follows (Line 6, Function 4). For each incoming motif edge to U with label d', such that $\langle V, j' \rangle \in \mathsf{E}'$ and where $d' \in (0, d]$, a motif edge with the same label d' is copied to $\mathscr{A}$, if and only if the number of set bits in the bit-vector of the potential motif edge to $\mathscr{A}$ is at least k (Lines 4-5, Function 11). If the bit-vector of the new

motif edge to $\mathscr{A}$ is equivalent to that of the corresponding motif edge to U, the duplicate motif edge to U is deleted (Lines 8-9, Function 11).

Finally, $\mathsf{isMotif}(\mathscr{A})$ is updated as described earlier (Line 13, Function 4 which calls Function 8).

Remark 3.4.1. *It is evident that as $\mathscr{A}$ moves down a branch in the suffix tree and $D(\mathscr{A})$ elongates, $|\mathsf{occ}(\mathscr{A})|$ reduces each time it passes an internal node. Thus, each seed node is augmented with a subset of edges of its parent seed node.*

The active point $\mathcal{A}$ eventually becomes an explicit node $V_{\mathcal{A}}$.

In this case, $\mathscr{A} = \langle V_{\mathscr{A}}, 0, i\rangle$, thus no extra work is being done by initialising $\mathscr{A}$ as a seed node prematurely. At this point, all information (including motif edges) is copied from $\mathscr{A}$ to $V_{\mathscr{A}}$ and a leaf node representing suffix j is added from $V_{\mathscr{A}}$. In preparation for the next step, E' is initialised as mentioned earlier. These steps correspond to Lines 8-11 in Function 3 and Function 12.

The active point $\mathcal{A}$ moves to another branch following the suffix link from $V_{\mathcal{A}}$.

After the leaf node j is added, $\mathscr{A}$ moves to a different branch by following the suffix link from $V_{\mathscr{A}}$ and becomes $\mathscr{A} = \langle s(V_{\mathscr{A}}), 0, i\rangle$ (Lines 11-12, Function 3). This is in order to prepare to add the leaf node representing the next suffix from $U = s(V_{\mathscr{A}})$.

Remark 3.4.2. *It is at this point that j is incremented by one (Line 13, Function 3).*

First, $|\mathsf{occ}(U)|$ is incremented by one and $\mathsf{isSeed}(U)$ is updated (Lines 3-4, Function 13). If $\mathsf{isSeed}(U) = \text{TRUE}$ and $|\mathsf{occ}(U)| \geq k$, then $\mathscr{B}(U)$ and $\mathsf{pivot}(U)$ are updated to reflect the

new occurrence of U at position j (Lines 5-6, Function 13). Then, in order to record the ending position of the occurrence of U starting at position j, a pointer to U is added to the list of pointers representing position $j+D(U)-1$ in E (Line 7, Function 13).

Motif edges are added as described earlier (Line 8, Function 13) but with one crucial difference: when a motif edge is added, the cluster to which it belongs is deleted from E' (Lines 9-10, Function 13). This is done in order to ensure left-maximality by preventing the addition of edges to suffixes of $V_{\mathscr{A}}$ that already exist to $V_{\mathscr{A}}$.

Lastly, $\mathsf{isMotif}(U)$ is updated (Line 11, Function 13).

An update is also done in the aforementioned way for all ancestors of U (Line 2, Function 13).

Remark 3.4.3. *It is at this point that the leaf $j-\ell$ (thus, the leftmost letter of the window) must be deleted (Lines 15-16, Function 3).*

There are two possibilities after this step (Line 17, Function 3):

- If another suffix link is to be followed due to the addition of the next suffix as a leaf node, then firstly, note that it is at this point that leaf $j-\ell$ must be deleted. Secondly, E' is updated to reflect the incrementation of j by deleting all pairs such that before the incrementation $j'=j-d-1$, and adding pairs obtained from E such that before the incrementation $j'=j-1$ (Function 14). This entire step is repeated until j is no longer incremented, and corresponds to Lines 17-23 in Function 3.
- There may come a point during the addition of subsequent suffixes as leaf nodes where $j \neq i$ and i once again begins increasing. In other words, $\mathscr{A}$ may follow suffix links around the suffix tree multiple times as j increases, but $\mathscr{A}$ may halt on a branch

and begin moving down it. As μ is incremented to 1 after $\mathscr{A} = \langle V, 0, i \rangle$ followed a suffix link to V, all ancestor seed nodes of V are updated, from the shallowest to the deepest, by simulating the movement of $\mathscr{A}$ down the current branch as described at the beginning of this section. This corresponds to Function 3 in its entirety.

3.4.2 Deleting a letter from the left

When the leaf node j is added, the leaf node $j - \ell$ must be deleted and its explicit parent node may also be deleted, as described in Section 3.3.2 (Lines 15-16 & 22-23 in Function 3). This section corresponds to Function 15, the function deleteLetter(j, ST), called on Line 2, updates the suffix tree and returns V', the deepest remaining ancestor node of the deleted leaf node, as well as the active point $\mathscr{A}$.

The following is done for each internal node V in the path from V' to the root, inclusive (Line 3 in Function 15). Note that, every node V is necessarily a seed node due to its left-maximality. The number of occurrences of V is decremented by one and isSeed(V) is updated accordingly (Lines 4-5 in 15). Outgoing motif edges are then updated as follows.

We say a motif edge $V \xrightarrow{d'} Z$ is *relevant* if it originates at V and its destination is a node Z which occurs at position $j - \ell + D(V) + d'$, where $d' \in (0, d]$. That is, $V \diamond^{d'} Z$ has lost one occurrence. After losing the occurrence, if $|\mathsf{occ}(V \diamond^{d'} Z)| = k - 1$, the edge $V \xrightarrow{d'} Z$ must be deleted. Alternatively, we say the edge is *affected*.

If $|\mathsf{occ}(V)| = k - 1$ or V is no longer a seed, all of its motif edges are outgoing, relevant and must be deleted (Lines 9-11 in Function 15). If $|\mathsf{occ}(V)| \geq k$, only a subset of its outgoing edges may be relevant and of those, a subset may be deleted and another subset may be affected. Relevant motif edges are dealt with depending on $s(V)$, the suffix target of V, as follows (Lines 16-17 in Function 15):

- If $s(V) \neq r$, any outgoing motif edge of V that is not also from $s(V)$, with the same label d' and destination node Z, is copied. If $|\mathsf{occ}(V \diamond^{d'} Z)| = k-1$, then the corresponding motif edge from V is deleted. This corresponds to Lines 18-19 in Function 15, and Function 21.
- If $s(V) = r$, any motif edge $V \xrightarrow{d'} Z$ such that $|\mathsf{occ}(V \diamond^{d'} Z)| = k-1$ is deleted, where $d' \in (0, d]$ and Z is the destination node. This corresponds to Lines 20-21 in Function 15.

Nodes V and $s(V)$ are then updated as follows.

- If $|\mathsf{occ}(V)| = k-1$ or V is no longer a seed, there are two possible cases:
  - If the non-empty suffix target $s(V)$ of V was not a seed, it becomes a seed and all information is moved from V to $s(V)$. This corresponds to Line 7 in Function 15 which calls Function 16.
  - If the non-empty suffix target of V was already a seed, then pointers to V in E, $\mathscr{B}(V)$, $\mathsf{pivot}(V)$ and $\mathsf{isMotif}(V)$ are all deleted (Lines 14-15 in Function 15).
- If $|\mathsf{occ}(V)| \geq k$, then $\mathsf{pivot}(V)$ and $\mathscr{B}(V)$ are updated (Line 22 in Function 15).

For every motif edge that is added from $s(V)$ by this update, $\mathsf{isMotif}()$ of $s(V)$ and of the corresponding destination nodes must be updated (Function 20). Finally, $\mathsf{isMotif}(V)$ is updated (Function 22).

Case 1 only.

If the active point $\mathscr{A}$ was pointing to a node U and node U was deleted as it had only one child remaining, then all information must be copied from node U to $\mathscr{A}$ prior to its deletion. This is handled by Function $\mathsf{deleteLetter}(j, \mathsf{ST})$, as defined in Section 3.4.2.

Case 2 only.

If the active point $\mathscr{A}$ was pointing to a node U and was moved to $s(U)$, observe that this equates to the case on the right-hand side of the window when $\mathscr{A}$ moves to a new branch. Thus, all internal nodes, as well as the information for $\mathscr{A}$, must be updated as described earlier in the current section. This corresponds to Lines 24-25 in Function 15.

3.4.3 Updating the Score Array

We are now in the position to describe how the algorithm computes the score array $\mathscr{S}_X$. Recall that each motif $\tilde{M}_{d,k} \in \mathscr{M}_{j,d,k}$ contributes to $\mathscr{S}_X[j]$ and $\tilde{M}_{d,k}$ has at least k occurrences in $X[j-\ell+1\,..\,j]$. A motif can either be a singleton motif or a maximal path of motif edges. A seed node U can be a singleton motif in any of the following cases:

- If U has no incoming or outgoing motif edges. That is, if $\mathsf{indegree}(U) = \mathsf{outdegree}(U) = 0$.
- If U does not have an incoming motif edge from a node V where each occurrence of U is preceded by V. That is, if $\nexists\, V \xrightarrow{d'} U$ such that $\mathscr{B}(U) \equiv \mathscr{B}(V \xrightarrow{d'} U)$.
- If U does not have an outgoing motif edge to a node V where each occurrence of U is succeeded by V. That is, if $\nexists\, U \xrightarrow{d'} V$ such that $\mathscr{B}(U) \equiv \mathscr{B}(U \xrightarrow{d'} V)$.

Before detailing how the score is updated for either side of the window, we describe how a maximal path can be elucidated on the right-hand side of the window, given a seed node U and a position j of one of its occurrences. A motif edge $V \xrightarrow{d'} U$, where V occurs at position $j-d'-D(V)$ and $\mathsf{kPopCount}(k, \mathscr{B}(V \xrightarrow{d'} U)) = \text{TRUE}$, can be extended to the left by following a path of motif edges, where at each extension, the following two conditions must hold:

- If $\mathsf{popCount}(\mathscr{B}(V \xrightarrow{d'} U)) = k$, the bit-vector of the extended path must match exactly with $\mathscr{B}(V \xrightarrow{d'} U)$. Alternatively, if $\textsc{popCount}(\mathscr{B}(V \xrightarrow{d'} U)) > k$, then the bit-vector of the extended path must contain at least k set bits, one of which must correspond to position j. In the latter case we make use of function $\mathsf{kPopCount}()$.

- The cumulative total of the labels d' of the motif edges in the path must not exceed d.

Similar logic applies when extending paths, to the right, on the left-hand side of the window.

Reading a Letter on the Right.

Following our earlier description of how updates are made to the motif graph, the score is updated for each relevant seed node U as follows:

- When a new motif edge $V \xrightarrow{d'} U$ is added, the score is incremented by one for each maximal path ending with $V \xrightarrow{d'} U$, if $V \xrightarrow{d'} U$ is not an extension of the prefix of the path. If no such paths exist, then the score is incremented by one for the motif edge. This corresponds to Lines 13-16 in Function 5, and Function 6.

- If $V \diamond^{d'} U$ gains an occurrence, the score is decremented by one for each maximal path ending with $V \xrightarrow{d'} U$, if $V \xrightarrow{d'} U$ is now an extension of the prefix of the path, such that the prefix already contributes to the score. This corresponds to Function 7.

- If a motif edge is duplicated due to $\mathscr{A}$ passing a seed node, the total score for all paths ending at the given motif edge is doubled (Lines 6-7 in Function 11). When $\mathscr{A}$ reaches a seed node and both share a motif edge from the same node with the same label, the total score for all paths ending at the given motif edge is halved (Lines 2-5 in Function 10).

- The score is incremented by one when a seed becomes a singleton motif; it is decremented by one when a singleton motif is no longer so (Function 8). The computation that establishes this has been described earlier and is done following the addition or deletion of any motif edges.

Deleting a Letter from the Left.

Following our earlier description of how updates are made to the motif graph, the score is updated for each relevant seed node V as follows:

- When a motif edge $V \xrightarrow{d'} Z$ from V must be deleted but $s(V) \xrightarrow{d'} Z$ already exists, the score is decremented by one for every maximal path that starts with $V \xrightarrow{d'} Z$ if its occurrence positions are distinct from those of the same paths but instead starting with $s(V) \xrightarrow{d'} Z$ (Function 18). Before $V \xrightarrow{d'} Z$ is deleted, the score is decremented by one for each path that begins with $V \xrightarrow{d'} Z$, if the occurrence positions of the suffix of the path differ (Lines 2-5 in Function 19). Then, the score is incremented by one for the suffix of the path, if it is not a prefix of a longer motif, which would already be contributing to the score (Lines 6-7 in Function 19).

- For each motif edge copied from V to $s(V)$, ending at some seed node Z, the score is incremented by one for each path starting with $s(V) \xrightarrow{d'} Z$ if it has different occurrence positions to $V \xrightarrow{d'} Z$ (Lines 11-13 in Function 21). If $V \xrightarrow{d'} Z$ has been deleted, the score is incremented by one without this check (Line 9 in Function 21). Alternatively, if $s(V)$ was already a seed and a singleton motif, and motif edge $s(V) \xrightarrow{d'} Z$ is added to it, if $s(V)$ has the same occurrence positions as $s(V) \diamond^{d'} Z$, $s(V)$ is no longer a singleton motif and the score is decremented by one. A similar check is done for the destination node Z whenever such a motif edge is added (Function 20).

- If $s(V)$ has just become a seed and a singleton motif, the score is incremented by one (Lines 2 & 6-7 in Function 16). If $|\mathsf{occ}(V)| = k-1$ or V is no longer a seed, but V was a singleton motif, the score is decremented by one (Lines 12-13 in Function 15). If V is still a seed, $|\mathsf{occ}(V)| \geq k$ and $\mathsf{isMotif}(V) = \text{FALSE}$ but all of its motif edges have been deleted, V becomes a singleton motif and the score is incremented by one (Lines 2-5 in Function 22). If any remaining outgoing motif edge from V has the same occurrence positions as V and V was a singleton motif, it is no longer so and the score is decremented by one (Lines 8-13 in Function 22). If no such match is found and V was not a singleton motif, it becomes one and the score is incremented by one (Lines 14-16 in Function 22).

3.5 Analysis

The following theorem summarises the complexity of the proposed algorithm.

Theorem 2. *Given a sequence X of length n, a window length ℓ, thresholds k and d, and size w of the machine word, the score array $\mathscr{S}_X$ is computed in $\mathscr{O}(nd\ell + d\lceil\frac{\ell}{w}\rceil \cdot \sum_{i=\ell}^{n-1} |\Delta_{i-1}^{i}|)$ time using $\mathscr{O}(\ell^2)$ space.*

Proof. Given a string X of length n, the suffix tree can be built for a sliding window in $\mathscr{O}(n)$ time using Ukkonen's [170] and Senft's [155] algorithms. Each time a leaf is added, the number of occurrences on all of its $\mathscr{O}(\ell)$ ancestor nodes must be incremented by one. This results in $\mathscr{O}(n\ell)$ time complexity for building and maintaining the nodes of the suffix tree. For each node V_i in the suffix tree, where $i < \ell$ per window, at most $\mathscr{O}(d\ell)$ motif edges can be added. Thus, in the worst case, the number of motif edges added are bounded by $\mathscr{O}(nd\ell)$. Each motif corresponds to at least one motif edge, so there cannot be more than $|\mathscr{M}_{i,d,k}|$ motif edges in the motif graph per window. The time required to update the motif graph at each deletion and addition step is proportional to the number of occurrences of motifs

deleted and introduced at either end of the window. All bit-operations used in the algorithm can be implemented in $\mathcal{O}(\lceil \frac{\ell}{w} \rceil)$ time. Thus elucidating each maximal path requires $\mathcal{O}(d\lceil \frac{\ell}{w} \rceil)$ time. When a leaf is added, updating E takes $\mathcal{O}(\ell)$ time. Thus the overall time complexity for maintaining E is $\mathcal{O}(n\ell)$. Summing the above gives the overall time complexity.

The size of the motif graph is $\mathcal{O}(\ell \cdot \lceil \frac{\ell}{w} \rceil)$, where the largest extra structure that each node holds is a bit-vector of size $\mathcal{O}(\lceil \frac{\ell}{w} \rceil)$. The lookup table used by popCount() is of size $\mathcal{O}(2^{\log_2 \ell}) = \mathcal{O}(\ell)$. The dynamic structure E of size $\mathcal{O}(\ell^2)$ gives an upper bound for the space complexity as each of the $\mathcal{O}(\ell)$ positions in E can hold pointers to at most ℓ seed nodes. The array $\mathcal{S}_X$ is not stored in memory as scores are reported in an on-line fashion. □

3.6 Final Remarks

In this section, we presented a motif discovery algorithm with the purpose of finding biologically significant regions in genomic sequences.

Our immediate target is to verify our theoretical findings and claims of improvement compared with the algorithm presented in [71] by implementing Algorithm MMDSW and testing it using synthetic and real genomic data.

In the future, an interesting possibility for the extension of this work could be in finding maximal motifs that occur a minimum number of times in each sequence given a set of multiple sequences, somewhat resembling the (ℓ, d)-motif search problem. Furthermore, our definition of maximal motifs could be extended to gapped maximal motifs. Results could also be refined by restricting the minimum length of a maximal motif, or adding a probabilistic postprocessing step.

Chapter 4

Pattern Matching in Pan-Genomes

4.1 Introduction

This chapter is based on work published in [67], which has been cited in 11 publications. The author's personal contribution to the work was in refinement to the design of the algorithm, as well as its implementation and experimentation.

4.1.1 Biological Motivation

It is possible to represent closely-related sequences, that have been aligned using a multiple sequence alignment (MSA) algorithm (defined in Section 1.3.2), into one compacted form that is able to represent the non-polymorphic sites (columns) of the MSA, as well as the polymorphic ones [81]. This representation compresses maximal sequences of non-polymorphic sites, while the polymorphic ones (containing substitutions, insertions, and deletions of letters) are represented as a set containing all possible variants observed at that location.

Example 29. *Consider the following MSA of three closely-related sequences, where mismatches are underlined:*

$$ATGCA\underline{A}C\underline{G}GG\underline{TA--}TTTTA$$

$$ATGCA\underline{A}C\underline{G}GG\underline{TATA}TTTTA$$

$$ATGCA\underline{C}C\underline{T}GG\underline{----}TTTTA$$

Thus, closely-related sequences can be compacted into a single string containing some deterministic and some non-deterministic segments. The latter are known as *degenerate segments*.

Definition 5. *A degenerate segment is a finite set of deterministic strings and may contain an empty string ε, corresponding to a deletion.*

Example 30. *The MSA from Example 29 can be compacted in to string $\tilde{T}$ as follows:*

$$\tilde{T} = \left\{ ATGCA \right\} \cdot \left\{ \begin{array}{c} A \\ C \end{array} \right\} \cdot \left\{ C \right\} \cdot \left\{ \begin{array}{c} G \\ T \end{array} \right\} \cdot \left\{ GG \right\} \cdot \left\{ \begin{array}{c} TA \\ TATA \\ \varepsilon \end{array} \right\} \cdot \left\{ TTTTA \right\}$$

Definition 6. *The total number of segments is the* length *of $\tilde{T}$ and the total number of letters is the* size *of $\tilde{T}$.*

This compact representation has been defined in [83] as an *elastic-degenerate* (ED) text. The natural problem that arises is finding all matches of a deterministic pattern P in text $\tilde{T}$. We call this the ELASTIC-DEGENERATE STRING MATCHING (*EDSM*) problem. The

simplest version of this problem assumes that a degenerate segment can contain only single letters [79].

An ED text can represent, for example, a set of closely-related DNA sequences. For instance, a *pan-genome* [38, 92, 118, 156, 166] is a reference sequence which is not simply a single genome, but the result of an MSA of several genomes that share large consensus regions and also exhibit mutations at some positions. Recently, various data structures to store pan-genomes have been suggested [13, 78].

4.1.2 Previous Work

Due to the application of cataloguing human genetic variation [167], there has been ample work in literature on the *off-line* (indexing) version of the pattern matching problem.

The authors of [107] initially proposed the extension of the FM-index [28, 58] compressed data structure to store and query ED strings. A similar approach termed the FM-index of alignment was presented in [114] and was shown to be less time-efficient in most pattern matching cases. In [81], the authors presented a novel concept for representing ED strings as follows. One string from each ED segment is present in the correct location within the string, and the remainder are appended to the genome, separated by unique symbols not in Σ and padded on both sides with bases flanking the corresponding original position in the genome. Making use of IUPAC-IUBMB symbols (introduced in Section 3.1.1) and the Gray code [66] to ensure efficient ordering of suffixes, the FM-index is constructed for the elongated genome. An extension of the backwards search algorithm [58] is then used for exact and approximate pattern matching. A similar approach was presented in [105] and shown to be less time-efficient.

Reference genomes and their variants can also be represented as graphs [1]. These graphs can be transformed in to compressed de Bruijn graphs [47] and then encoded using FM-index-based approaches [159, 160] which can be queried in a manner similar to the above solutions.

Furthermore, several approaches are based on Lempel-Ziv compression, which is one of the most commonly used compression algorithms [50, 57, 95, 96, 182].

In literature, there are also algorithms and applications for the problem of inferring motifs from degenerate input texts [131, 145]. However, to the best of our knowledge, the on-line, more fundamental, version of the EDSM problem has not been studied as much as indexing approaches. Solutions to the on-line version can be beneficial for a number of reasons:

- efficient on-line solutions can be used in combination with partial indexes as practical trade-offs;
- efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching, similar to standard strings [12];
- on-line solutions can be useful when one wants to search for a few patterns in many degenerate texts, similar to standard strings [2].

Let us denote by m the length of pattern P; by n the length of $\tilde{T}$; and by $N > m$ the size of $\tilde{T}$. In [83], an algorithm for solving the *EDSM* problem in time $\mathcal{O}(\alpha\gamma mn + N)$ and space $\mathcal{O}(N)$ was presented; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate segment of the text and the maximum number of degenerate segments spanned by any occurrence of the pattern in the text. The algorithm uses the Knuth-Morris-Pratt (KMP) algorithm (see Section 4.2 for a summary) to find what are defined as:

- Type 1 occurrences of P in deterministic segments of $\tilde{T}$;
- Type 2 occurrences of P in the strings in degenerate segments of $\tilde{T}$;
- Type 3 occurrences of prefixes of P in either deterministic or degenerate segments of $\tilde{T}$.

Definition 7. *The lowest common ancestor (LCA) of a pair of nodes in a suffix tree is the deepest node which is an ancestor of both nodes. An LCA query can be computed for any pair of nodes in constant time, following linear time and space pre-processing [152].*

Type 3 occurrences are then extended using LCA queries on either of the two generalised suffix trees built on:

- P concatenated with all deterministic segments of $\tilde{T}$;
- P concatenated with all strings in all degenerate segments of $\tilde{T}$.

4.1.3 Chapter Summary

In this chapter, we improve the state of the art. Specifically, our contribution is twofold.

Algorithm. In Section 4.3, we present an algorithm to solve the EDSM problem in an on-line manner which requires time $\mathcal{O}(nm^2+N)$, after a preprocessing stage with time and space $\mathcal{O}(m)$.

Experimental results. In Section 4.5, we present experiments confirming our theoretical findings in practical terms. We also compare the presented algorithm to other algorithms solving the same problem, including the one described in Section 4.1.2.

4.2 Definitions

By $\mathsf{PrefSuf}_{U,V}$ we denote the set containing all indices i, such that the prefix $U[0..i]$ of string U is also a suffix of string V.

Example 31. *Suppose we have two strings $U = \texttt{ATATG}$ and $V = \texttt{CATAT}$. Then $\mathsf{PrefSuf}_{U,V} = \{1,3\}$ because of prefixes/suffixes $\texttt{AT}$ and $\texttt{ATAT}$, respectively.*

An *elastic-degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1]\cdots\tilde{X}[n-1]$, of *length* n, on an alphabet Σ, is a finite sequence of n *elastic-degenerate letters*. Every *elastic-degenerate letter* $\tilde{X}[i]$, for all $i \in [0,n)$, is a non-empty set of strings $\tilde{X}[i][j]$, where $j \in [0,|\tilde{X}[i]|)$ and each $\tilde{X}[i][j]$ is a deterministic string on Σ. The total *size* N of $\tilde{X}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

Only in this chapter and for the purpose of computing N, we define $|\varepsilon| = 1$.

Remark 4.2.1. *For an ED string $\tilde{X}$, the size and the length are two distinct concepts. See Example 32 below.*

Example 32 (Running example)**.** *Suppose we have an ED string $\tilde{T}$ as shown below. The length of $\tilde{T}$ is $n = 6$ and its size is $N = 18$.*

$$\tilde{T} = \left\{ \texttt{C} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{C} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{AC} \\ \texttt{ACC} \\ \texttt{CACA} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{C} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{AC} \end{matrix} \right\} \cdot \left\{ \texttt{C} \right\}$$

We say that a deterministic string Y *matches* an ED string $\tilde{X} = \tilde{X}[0]..\tilde{X}[m'-1]$, of length $m' > 1$, denoted by $Y \approx \tilde{X}$, if and only if string Y can be decomposed into $y_0 \dots y_{m'-1}$, where $y_i \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $i \in [1, m'-2]$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m'-1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that in the above definition we require that both y_0 and $y_{m'-1}$ are non-empty to avoid degenerate cases at the beginning or at the end of an occurrence. A deterministic string Y, of length m, is said have an *occurrence* ending at position j in an ED string $\tilde{T}$ if there exist $i < j$ such that $\tilde{T}[i]..\tilde{T}[j] \approx Y$, or, if there exists $s \in \tilde{T}[j]$ such that Y occurs in s.

Example 33 (Running example)**.** *Given a deterministic pattern $P =$ `ACACA`, of length $m = 5$, and $\tilde{T}$ from Example 32, the first occurrence of P starts at position* 1 *and ends at position* 2 *of $\tilde{T}$ and the second occurrence starts at position* 2 *and ends at position* 4*. The second occurrence is found due to the existence of the suffix* `ACA` *of the third string* `CACA` *in position* 2*; the first string* `C` *in position* 3*; and either the first string* `A` *or the prefix* `A` *of the second string* `AC` *in position* 4*.*

We are now in a position to formally define the main problem of this chapter.

ELASTIC-DEGENERATE STRING MATCHING (*EDSM*)
Input: a deterministic string P, of length m, and an ED string $\tilde{T}$, of length n and total size $N \geq m$.
Output: all positions j in $\tilde{T}$ where at least one occurrence of P ends.

Algorithmic Tools

Fact 4 ([54]). *Finding all $|occ(X)|$ occurrences of a string X, of length m, in a string Y, of length, $n > 0$, can be performed in time $\mathscr{O}(m + |occ(X)|)$ using ST_Y.*

A *border* of a non-empty string X is a proper factor of X that is both a prefix and a suffix of X. We introduce the function $\mathsf{border}(X)$ defined for a non-empty string X as the longest border of X. Let X be a string of length $m \geq 1$. We define the *border table* $\beta : \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$ by $\beta[k] = |\mathsf{border}(X[0..k])|$, for $k \in [0,m)$.

Fact 5 ([44]). *Given a string X, of length m, the border table of X can be computed on-line in time $\mathscr{O}(m)$. All borders of X can be specified in time $\mathscr{O}(m)$ using the border table.*

Remark 4.2.2. *The border table and the notion of a border refer to a proper prefix and a proper suffix of the same string, whereas the indexes in $\mathsf{PrefSuf}_{X,Y}$ refer to a string which is a prefix of string X and a suffix of another string Y, which is not necessarily proper.*

Lemma 5. *Given a string X, of length m, and the suffix tree ST_Y of a string Y, of length n, $\mathsf{PrefSuf}_{X,Y}$ can be computed in time $\mathscr{O}(m)$.*

Proof. By applying Fact 4, we traverse ST_Y to find the terminal node V corresponding to the longest prefix of X which is path-labelled $\mathscr{P}(V)$. While traversing ST_Y with X, we add index $n-1-i$ to $\mathsf{PrefSuf}_{X,Y}$ if we encounter a terminal node U, such that $\mathscr{P}(U) = Y[i..n-1]$. The longest such prefix of X is of length at most m. No longer prefix of X can be a suffix of Y as it does not occur in Y. □

Knuth-Morris-Pratt (KMP) Algorithm

This summary is based on [94] where further details may be found. The KMP algorithm allows the efficient search of a pattern P, of length m, in a text, of length n. A pre-processing step computes the border table of P in time $\mathcal{O}(m)$. The text is then searched in $\mathcal{O}(n)$ time, and at any position at which a mismatch occurs, the border table is used to ensure that characters of the text at which there could not possibly be a match are skipped. The overall time complexity is therefore $\mathcal{O}(n+m)$.

4.3 Algorithm

An ED string represents an exponential number of strings per ending position, where the exact number is the product of the number of deterministic strings at previous positions. Searching a pattern in all these strings naïvely is thus not acceptable.

Main Idea

The pseudocode for our algorithm is presented later as Algorithm EDSM and is referred to throughout this section. The algorithm has a preprocessing phase where the suffix tree ST_P of the pattern P is built (Line 2 in Algorithm EDSM). Then, in an on-line manner, we scan $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$, we:

1. memorise the prefixes of the pattern that occur at the end of $\tilde{T}[i]$ (Lines 6 & 14);
2. check whether at $\tilde{T}[i]$ it is possible to extend an occurrence of the pattern which has started earlier in $\tilde{T}$ (Lines $15-18$);
3. in both previous cases we finally check whether such an occurrence of P actually also ends in $\tilde{T}[i]$ (Lines $7-9$ & $19-24$).

We perform these steps by computing and storing the list $\mathscr{L}_i$ of the rightmost positions of prefixes of P that occur at the end of $\tilde{T}[i]$, for all i where $i \in [0,n)$.

Algorithm EDSM

Below, we formally present Algorithm EDSM that solves Problem EDSM (defined in Section 4.2) in an on-line manner. Note that by $\mathsf{insert}(A,\mathscr{L})$, we denote the operation that inserts the elements of a set A into a linked-list $\mathscr{L}$.

```
1  Algorithm EDSM(P, m, T̃, n)
2      construct ST_P;
3      ℒ_0 ← EmptyList();
4      foreach S ∈ T̃[0] do
5          compute PrefSuf_{P,S} using the border table;
6          insert(PrefSuf_{P,S}, ℒ_0);
7          if |S| ≥ m then
8              search P in S using KMP and
9              report 0 if P occurs in S and checkDuplicate(0);
10     foreach i ← 1 to n − 1 do
11         ℒ_i ← EmptyList();
12         foreach S ∈ T̃[i] do
13             compute PrefSuf_{P,S} using border table;
14             insert(PrefSuf_{P,S}, ℒ_i);
15             if |S| < m then
16                 𝒜 ← search S in P using ST_P;
                   ▷ denote starting positions by 𝒜
17                 foreach (p ∈ ℒ_{i−1}, j ∈ 𝒜) s.t. p + 1 = j do
18                     insert({p + |S|}, ℒ_i);
19             if |S| ≥ m then
20                 Search P in S using KMP and
21                 report i if P occurs in S and checkDuplicate(i);
22             Compute PrefSuf_{S,P} using ST_P;
23             if ∃(p ∈ ℒ_{i−1}, j ∈ PrefSuf_{S,P}) s.t. p + j + 2 = m then
24                 Report i if checkDuplicate(i);
```

Example 34 below shows steps (1) and (2) on a running example. The border table shown in Example 34 has to be computed for all positions of the text, leading to the overall complexity stated in Lemma 6.

Example 34 (Running example)**.** *Let us consider again P and $\tilde{T}$ as before. Assume we have already computed $\mathscr{L}_0$ and $\mathscr{L}_1$, and we move to position $i = 2$, where at $\tilde{T}[i]$ we have three strings $\{S_0, S_1, S_2\}$, with $S_0 = \texttt{AC}$, $S_1 = \texttt{ACC}$ and $S_2 = \texttt{CACA}$. We generate the string*

$$X_i = X_2 = P\,\$_0\,S_0\,\$_1\,S_1\,\$_2\,S_2 = \texttt{ACACA}\$_0\,\texttt{AC}\$_1\,\texttt{ACC}\$_2\,\texttt{CACA}$$

and build its border table β (Line 13*).*

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X_2[k]$	A	C	A	C	A	$\$_0$	A	C	$\$_1$	A	C	C	$\$_2$	C	A	C	A
$\beta[k]$	0	0	1	2	3	0	1	2	0	1	2	0	0	0	1	2	3

In order to compute $\mathsf{PrefSuf}_{P,S}$ (Line 13*), we read $\beta[7] = 2$, which gives the length of the longest string which is a prefix of P and a suffix of S_0. We check if there exist borders of length shorter than* 2*. We read $\beta[2-1] = 0$, telling us that no shorter border exists; so we have $\mathsf{PrefSuf}_{P,S_0} = \{1\}$. We then read $\beta[11] = 0$, telling us that no prefix of P is a suffix of S_1, and hence $\mathsf{PrefSuf}_{P,S_1} = \emptyset$. We read $\beta[16] = 3$, which gives the length of the longest string which is a prefix of P and a suffix of S_2. We check if there exist shorter borders: we read $\beta[3-1] = 1$, telling us that a shorter border of length* 1 *exists. Since $\beta[1-1] = 0$, no shorter border exists. So we have $\mathsf{PrefSuf}_{P,S_2} = \{0,2\}$. This gives us a partial $\mathscr{L}_i = \{0,1,2\}$ for position $i = 2$ that concludes Step* (1) *for position $i = 2$ (Line* 13*).*

Further on, at Step (2) *we will add position* 4 *to $\mathscr{L}_2$ by extending the occurrence of P that had started at $\tilde{T}[1]$. Putting everything together, we get $\mathscr{L}_2 = \{0,1,2,4\}$ (Lines* $17-18$*).*

Lemma 6. *Given P, of length m, and $\tilde{T}$, of length n and size N, the sets $\mathsf{PrefSuf}_{P,S}$ where $S \in \tilde{T}[i]$, for all $i \in [0,n)$, can be computed in time $\mathcal{O}(N)$.*

Proof. For each position i, we generate a string $X_i = P\$_0 S_0 \$_1 S_1 \$_2 S_2 \ldots \$_{k-1} S_{k-1}$, where $S_j \in \tilde{T}[i]$ and each $\$_j$ is a distinct letter not in Σ, where $j \in [0,k)$. We build the border table β of string X. By traversing β from left to right we can compute sets $\mathsf{PrefSuf}_{P,S_j}$. Specifically, for any string S_j, all borders that are suffixes of S_j and prefixes of P can be computed in time $\mathcal{O}(|S_j|)$ since there exists at most $|S_j|$ such borders. By Fact 5, we can build all border tables, and hence compute all $\mathsf{PrefSuf}_{P,S_j}$ for all $S_j \in \tilde{T}[i]$ in time

$$\mathcal{O}(|P| + \sum_{j=0}^{k-1} |S_j|).$$

Since the length and the total size of $\tilde{T}$ is n and N, respectively, sets $\mathsf{PrefSuf}_{P,S_j}$ can be computed in time $\mathcal{O}(nm+N)$. By noting that the border table for P can be computed only once and that the border table computation can be done online (Fact 5), the whole computation is bounded by $\mathcal{O}(N)$. □

Lemma 7. *Given P, ST_P, and $\tilde{T}$ of length n and size N, the sets $\mathsf{PrefSuf}_{S,P}$, $S \in \tilde{T}[i]$, for all $i \in [1,n)$, can be computed in time $\mathcal{O}(N)$.*

Proof. By Lemma 5, for any $S \in \tilde{T}[i]$, if $|S| \leq |P|$, $\mathsf{PrefSuf}_{S,P}$ can be computed in time $\mathcal{O}(|S|)$ using ST_P. Since the total size of $\tilde{T}$ is N, all sets $\mathsf{PrefSuf}_{S,P}$ can be computed in time $\mathcal{O}(N)$. □

Lemma 8. *Lists $\mathscr{L}_i$, for all $i \in [0,n)$, in Algorithm EDSM can be computed in time $\mathcal{O}(nm^2 + N)$.*

Proof. List $\mathscr{L}_0$ consists of the elements of $\mathsf{PrefSuf}_{P,S}$ for position 0, which by Lemma 6 can be done within time $\mathscr{O}(N)$. For pattern P, of length m, there exist at most $\frac{m(m+1)}{2}$ factors. For the strings $S_j \in \tilde{T}[i]$, where $|S_j| \leq m$ and $j \in [0,k)$, we can find at most $\frac{m(m+1)}{2} = \mathscr{O}(m^2)$ occurrences in pattern P. By Fact 4, finding all occurrences can be done in time

$$\sum_{j=0}^{k-1} (|S_j| + |\mathsf{occ}(S_j)|)$$

and this is bounded by $\mathscr{O}(nm^2+N)$ for all positions i. This is because, by definition, no $S_j, S_{j'} \in \tilde{T}[i]$ exist such that $S_j = S_{j'}$. Each occurrence can cause only one extension from $\mathscr{L}_{i-1}$ to $\mathscr{L}_i$. To avoid duplicates in $\mathscr{L}_i$, we need to check if there exist multiple prefix extensions ending at the same position. Each check can be done in constant time using a bit vector of size m, which we set on only once per position i of $\tilde{T}$. Therefore, we can extend the prefixes in time $\mathscr{O}(m^2)$ for each position i, and in time $\mathscr{O}(nm^2)$ for the whole text $\tilde{T}$ of length n, where $i \in [0,n)$. By Lemma 6, sets $\mathsf{PrefSuf}_{P,S}$ corresponding to new prefixes of pattern P which are suffixes of $\{S_0, S_1, \ldots, S_{k-1}\}$ at position $\tilde{T}[i]$ can be found in time $\mathscr{O}(N)$. Merging new prefixes and the prefixes extended from $\mathscr{L}_{i-1}$ can be done in time $\mathscr{O}(m)$ since both are at most m. Therefore, lists $\mathscr{L}_i$, for all $i \in [0,n)$, in EDSM can be computed in time $\mathscr{O}(nm^2+N)$. □

Example 35 below shows Step 3 on our running example.

Example 35 (Running example)**.** *Let us consider again P and $\tilde{T}$ as before. At the step $i = 4$, we have $\mathscr{L}_3 = \{1,3\}$ and we have to compute $\mathscr{L}_4$. For $S_0 = \texttt{A}$, we have $\mathsf{PrefSuf}_{\texttt{A},\texttt{ACACA}} = \{0\}$ (Line 22), so for $3 \in \mathscr{L}_3$, we have that $3+0+2 = 5 = m$ (Line 23). Hence, one occurrence of P has been found. Moreover, for $S_1 = \texttt{AC}$, we have $\mathsf{PrefSuf}_{\texttt{AC},\texttt{ACACA}} = \{0,1\}$ (Line 22), so for $3 \in \mathscr{L}_3$, we have that $3+0+2 = 5 = m$ (Line 23). Therefore, another occurrence of P has been found at the same position.*

Since our algorithm reports all positions i in $\tilde{T}$ where at least one occurrence of P ends, and since more than one occurrence may end at the same position (as in Example 35), we need to avoid duplications. To this end, we can use a simple operation $\mathsf{checkDuplicate}(i)$ to check whether the current position i has already been reported (Lines 9, 21 & 24).

4.4 Analysis

Theorem 3. *Algorithm EDSM solves Problem* EDSM *in an on-line manner in $\mathscr{O}(nm^2+N)$ time. Algorithm EDSM requires $\mathscr{O}(m)$ preprocessing time and space.*

Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [94] if $|S| > m$ (where $S \in \tilde{T}[i]$), and from the combination of Lemmas 7 and 8, if $|S| \leq m$. By definition, we cannot have any other type of (ending) occurrence. By Fact 2, the suffix tree ST_P can be computed in $\mathscr{O}(m)$ time and space. By Lemma 8, lists $\mathscr{L}_i$, for all $i \in [0,n)$, can be computed in $\mathscr{O}(N+nm^2)$ time. By Lemma 7, sets $\mathsf{PrefSuf}_{S,P}$ can be computed in $\mathscr{O}(N)$ time. In case $|S| < m$, we use $\mathscr{L}_{i-1}$ and set $\mathsf{PrefSuf}_{S,P}$ to find and report occurrence i in $\mathscr{O}(m)$ time using a bit-vector of size m, which we initialise only once per position i. Finally, searching P in $S \in \tilde{T}[i]$, in case $|S| \geq m$, can be done in $\mathscr{O}(|S|)$ time using the KMP algorithm [94], which is bounded by $\mathscr{O}(N)$ for $\tilde{T}$ of total size N. The algorithm reads a position i and reports whether i is an ending position of some occurrence of P before reading position $i+1$. Therefore, Algorithm EDSM solves the *EDSM* problem in an on-line manner in $\mathscr{O}(nm^2+N)$ time, with $\mathscr{O}(m)$ preprocessing time and space. □

For some special cases, Algorithm EDSM can run faster.

Definition 8. *A set S of strings is* prefix-free *if no two distinct strings in S exist such that one is prefix of the other. We say that an ED string $\tilde{X}$ of length n is* locally prefix-free *if each set $\tilde{X}[i]$ is prefix-free, for $i \in [0,n)$.*

Corollary 1. For any locally prefix-free ED string, Algorithm EDSM solves the EDSM problem in an on-line manner in time $\mathcal{O}(nm+N)$.

4.5 Experimental Results

We have implemented Algorithm EDSM as program EDSM in the C++ programming language, available at `https://github.com/webmasterar/edsm` under the GNU General Public License. The implementation of the algorithm presented in [83], denoted by IKP, was obtained from `https://github.com/Ritu-Kundu/ElDeS`. We also obtained the implementation of Algorithm EDSM-BV, an algorithm developed later, that was an improvement of Algorithm EDSM and is summarised in the following. Algorithms IKP and EDSM-BV were chosen as they were the state of the art.

Algorithm EDSM-BV [67] is a *non-trivial* bit-vector version of Algorithm EDSM. The main idea of this algorithm is to simulate Algorithm EDSM using bit-level operations to maintain linked-lists $\mathscr{L}$ and do the matching. A further pre-processing step is also added to the suffix tree of the pattern. This augmented suffix tree allows for the retrieval of a bit-vector representation of all occurrences of a string $S \in \tilde{T}[i]$ in P in $\mathcal{O}(|S|)$ time. With this structure, bit-level operations can be used to compute $\mathscr{L}_i$ from $\mathscr{L}_{i-1}$. Algorithm EDSM-BV requires time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$, after a preprocessing stage with time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where w is the size of the machine word. Therefore, for short patterns, this algorithm requires time linear in

the size of the text. The implementation of this algorithm, program EDSM-BV is available at `https://github.com/webmasterar/edsm` under the GNU General Public License.

All three programs were compiled with g++ version 4.7.3 at optimisation level 3 (-O3). The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. We compared the performance of EDSM, EDSM-BV, and IKP using synthetic data; as well as the performance of EDSM-BV (shown to be the fastest) using real data. The synthetic datasets referred to in this section are maintained at `https://github.com/webmasterar/edsm`.

4.5.1 Time Performance

Synthetic ED strings were created arbitrarily (uniform distribution), with n ranging from $100,000$ to $1,600,000$ and the percentage of elastic-degenerate positions set to 10%. For each degenerate position within the synthetic ED strings, the number of strings was chosen arbitrarily, with an upper bound set to 10. The length of each string inside a segment was chosen arbitrarily, with an upper bound again set to 10. Four different patterns of length $m = 8, 16, 32$ or 64 were given as input to all three algorithms, along with the aforementioned synthetic ED strings, resulting in four sets of output shown in Figure 4.5.

Our theoretical findings, showing that EDSM and EDSM-BV are asymptotically faster than IKP, are validated in practice by the results illustrated in Figure 4.5. Note that the axes are in $\log_2$ scale. In particular, the results confirm that EDSM-BV, which is asymptotically the fastest for short patterns, is also the fastest in practice by up to two orders of magnitude. As for Algorithm EDSM, not surprisingly, we observe that, as m grows, the m^2 factor in its time complexity becomes increasingly significant overall. Note that searching for even longer

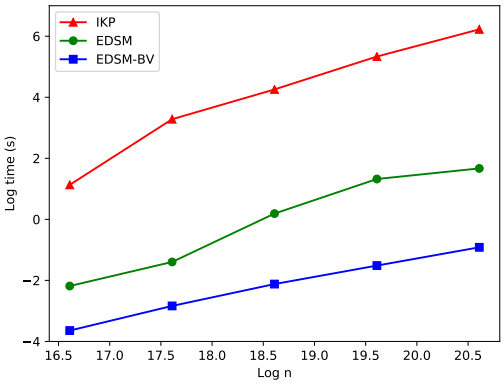


Figure 4.1 Pattern of length $m = 8$

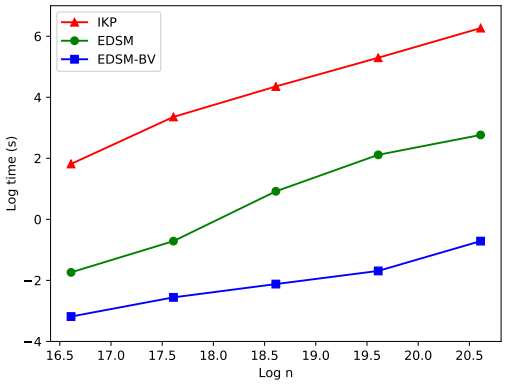


Figure 4.2 Pattern of length $m = 16$

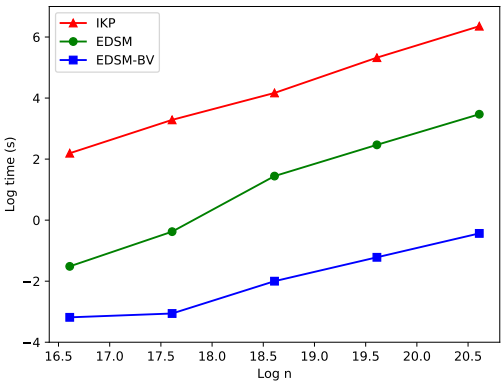


Figure 4.3 Pattern of length $m = 32$

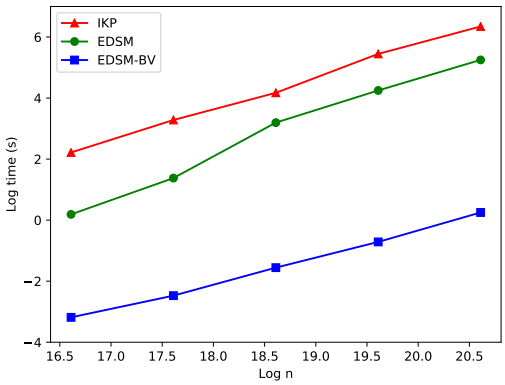


Figure 4.4 Pattern of length $m = 64$

Figure 4.5 Elapsed time of EDSM, EDSM-BV, and IKP for synthetic ED texts of length n.

patterns *exactly* is not relevant in bioinformatics, where errors (substitution, insertion, and deletion mutations) need to be accommodated as m grows.

4.5.2 Application to Real Data

Program EDSM-BV was tested further using real-world datasets. Human genomic data was obtained from the 1,000 Genomes Project [167]. Specifically, data was obtained from Phase 3 of the project, in which the genomes of 2,504 individuals from 26 different populations were sequenced and aligned, producing a dataset which summarises the variation in the

sample population. Files in Variant Call Format (VCF) include information about variations at each position in the reference genome, which makes the format ideal for our purposes. Tool EDSM-BV was given a reference sequence (FASTA format) and variation data (VCF) for each of the ten smallest chromosomes as input, as well as synthetic, randomly generated patterns of length $m = 8, 16, 32$ or 64. The processing time of EDSM-BV was recorded; by *processing* we refer only to the actual CPU time used in executing the process, excluding the time to read the data in memory on-line. Chromosome 21, which is the smallest in length, has a VCF file of size 11.2GB. The results of this experiment are displayed in Figure 4.10.

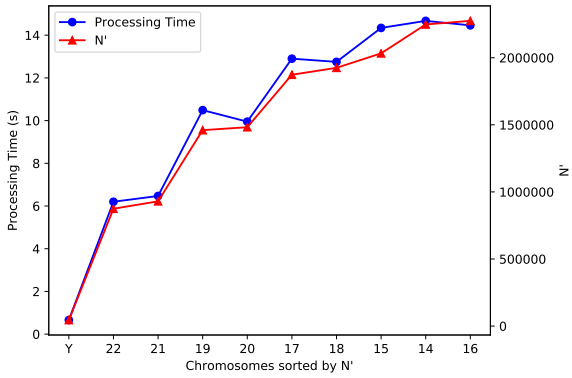


Figure 4.6 Pattern of length $m = 8$

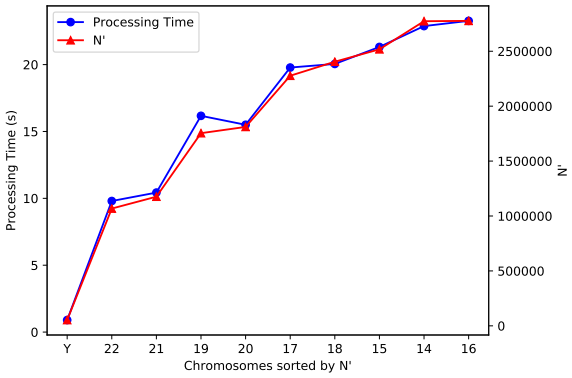


Figure 4.7 Pattern of length $m = 16$

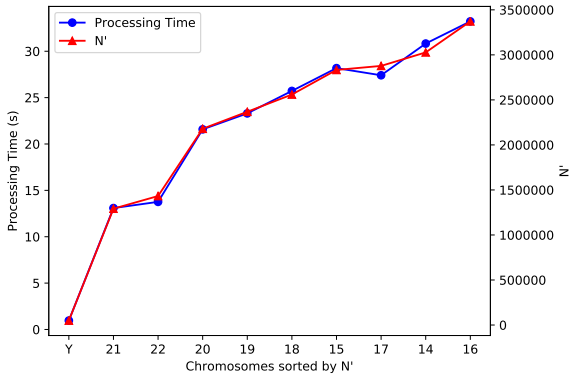


Figure 4.8 Pattern of length $m = 32$

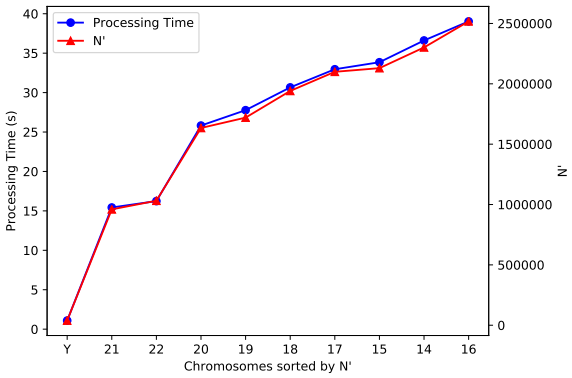


Figure 4.9 Pattern of length $m = 64$

Figure 4.10 Processing time of EDSM-BV for human genomic data.

The graphs in Figure 4.10 show, for the ten smallest chromosomes, a very clear *linear* relationship between the time taken for EDSM-BV to run and N', the total number of strings $S \in \tilde{T}[i]$ such that $|S| < |P|$ per chromosome.

4.6 Final Remarks

We have presented an efficient algorithm, Algorithm EDSM, for on-line pattern matching on a set of similar texts, such as genomes of the same or related species. Also, the presented experimental results confirm our theoretical findings in practical terms.

This work was published in [67], along with Algorithm EDSM-BV, and has been cited in several publications since:

- Considering the approximate version of Problem EDSM under the edit (resp. Hamming distance) model, in [19], the authors present a $\mathcal{O}(k^2mG + kN)$-time (resp. $\mathcal{O}(kmG + kN)$) and $\mathcal{O}(m)$-space algorithm, where k is maximum numbers of errors allowed and G is the total number of strings in $\tilde{T}$.

- An improvement on the time complexity of Algorithm EDSM was presented in [4], where the presented algorithm runs in $\mathcal{O}(nm\sqrt{m \log m} + N)$ time.

- As presented in [36], the same time complexity as Algorithm EDSM-BV can be achieved using bit-parallelism.

- Algorithm EDSM-BV has been adapted and improved for the purpose of dictionary matching, as presented in [133]. The presented algorithm runs in $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ time, with preprocessing in $\mathcal{O}(M)$ time and space, where M is the total length of all patterns.

Chapter 5

Conclusion

5.1 Thesis Summary

The rapidly advancing field of molecular sequence analysis presents innumerable opportunities for research, some of which have been addressed in this thesis. Below, we provide a summary of our results.

In Chapter 2, we presented the $\mathcal{O}(\beta m + n + L^3)$-time Algorithm saCSCr to solve the problem of correctly aligning two linear sequences (of length m and $n \geq m$) obtained from circular sequences. Recall that β is the number of blocks that the sequences are divided in to and L is the length of prefixes and suffixes of each sequence upon which the refinement step is performed in order to solve the problem exactly.

In Chapter 3, we presented Algorithm MMDSW to find interesting regions within genomes by finding maximal motifs. The algorithm works in

$$\mathcal{O}(nd\ell + d\lceil\frac{\ell}{w}\rceil \cdot \sum_{i=\ell}^{n-1} |\mathsf{diff}_{i-1}^{i}|)$$

time, where n is the length of the input text X; d is the upper-bound for the number of allowed degenerate symbols in any motif $\tilde{M}_{i,d,k}$; ℓ is the length of the window; w is the size of the machine word; and diff_{i-1}^{i} is the symmetric difference of the sets of occurrences of maximal motifs at $X[i-\ell\,..\,i-1]$ and at $X[i-\ell+1\,..\,i]$.

Finally, in Chapter 4, we present Algorithm EDSM to find the occurrences of a pattern, of length m, in an elastic-degenerate text, of length n and size N. The algorithm runs in $\mathcal{O}(nm^2+N)$ time and requires a $\mathcal{O}(m)$-time preprocessing step.

This thesis is based upon research presented in five publications [15, 67–69, 85] which, in total, have been extended, improved or cited 18 times.

5.2 Future Work

As well as the specific extensions of the presented work mentioned earlier, we provide below a list of bioinformatic applications of the presented algorithms:

- An experimental study of all UniProt protein sequences to discover novel pairs of circularly-permuted proteins using Algorithm saCSCr presented in Chapter 2.
- An experimental study of bacterial genomes from GenBank to predict the locus of OriC using Algorithm MMDSW presented in Chapter 3, and compare the results to those of OriFinder [104], a tool developed specifically to predict OriC using probabilistic methods.
- Algorithm EDSM presented in Chapter 4 can be utilised to align reads from newly-sequenced genomes to related pan-genomes.

References

[1] (2016). Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135.

[2] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410.

[3] Ameyar, M., Wisniewska, M., and Weitzman, J. B. (2003). A role for AP-1 in apoptosis: the case for and against. *Biochimie*, 85(8):747–52.

[4] Aoyama, K., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M., et al. (2018). Faster online elastic degenerate string matching. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 105. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] Apostolico, A., Comin, M., and Parida, L. (2010). Varun: discovering extensible motifs under saturation constraints. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(4):752–762.

[6] Apostolico, A., Pizzi, C., and Ukkonen, E. (2011). Efficient algorithms for the discovery of gapped factors. *Algorithms for Molecular Biology*, 6(1):5.

[7] Arimura, H. and Uno, T. (2007). An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *Journal of combinatorial optimization*, 13(3):243–262.

[8] Athar, T., Barton, C., Bland, W., Gao, J., Iliopoulos, C. S., Liu, C., and Pissis, S. P. (2015). Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, FirstView:1–14.

[9] Ayad, L. A., Barton, C., and Pissis, S. P. (2017). A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87.

[10] Ayad, L. A. and Pissis, S. P. (2017). Mars: improving multiple circular sequence alignment using refined sequences. *BMC genomics*, 18(1):86.

[11] Azim, M. A. R., Kabir, M., and Rahman, M. S. (2018). A simple, fast, filter-based algorithm for circular sequence comparison. In *International Workshop on Algorithms and Computation*, pages 183–194. Springer.

[12] Baeza-Yates, R. A. and Perleberg, C. H. (1996). Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27.

[13] Baier, U., Beller, T., and Ohlebusch, E. (2016). Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, 32(4):497–504.

[14] Bailey, T. L., Elkan, C., et al. (1994). Fitting a mixture model by expectation maximization to discover motifs in bipolymers.

[15] Barton, C., Iliopoulos, C. S., Kundu, R., Pissis, S. P., Retha, A., and Vayani, F. (2015a). Accurate and efficient methods to improve multiple circular sequence alignment. In *14th SEA*, volume 9125 of *LNCS*, pages 247–258.

[16] Barton, C., Iliopoulos, C. S., and Pissis, S. P. (2014). Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9:1–10.

[17] Barton, C., Iliopoulos, C. S., and Pissis, S. P. (2015b). Average-case optimal approximate circular string matching. In Dediu, A. H., Formenti, E., Martin-Vide, C., and Truthe, B., editors, *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 85–96. Springer.

[18] Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., Rapp, B. A., and Wheeler, D. L. (2000). GenBank. *Nucleic Acids Res*, 28(1):15–18.

[19] Bernardini, G., Pisanti, N., Pissis, S. P., and Rosone, G. (2017). Pattern matching on elastic-degenerate text with errors. In *International Symposium on String Processing and Information Retrieval*, pages 74–90. Springer.

[20] Boore, J. L. and Brown, W. M. (1998). Big trees from little genomes: mitochondrial gene order as a phylogenetic tool. *Current opinion in genetics & development*, 8(6):668–674.

[21] Bray, N. and Pachter, L. (2004). MAVID: constrained ancestral alignment of multiple sequences. *Genome Res*, 14(4):693–699.

[22] Brazma, A., Jonassen, I., Eidhammer, I., and Gilbert, D. (1998). Approaches to the automatic discovery of patterns in biosequences. *Journal of computational biology*, 5(2):279–305.

[23] Brodie, R., Smith, A. J., Roper, R. L., Tcherepanov, V., and Upton, C. (2004). Base-By-Base: Single nucleotide-level analysis of whole viral genome alignments. *BMC Bioinform*, 5(1):96.

[24] Buhler, J. and Tompa, M. (2002). Finding motifs using random projections. *Journal of computational biology*, 9(2):225–242.

[25] Bunke, H. and Buhler, U. (1993). Applications of approximate string matching to 2D shape recognition. *Pattern Recognit*, 26(12):1797–1812.

[26] Burcsi, P., Cicalese, F., Fici, G., and Lipták, Z. (2012). Algorithms for jumbled pattern matching in strings. *Int J Found Comput Sci*, 23(2):357–374.

[27] Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H.-P., Rivals, E., and Vingron, M. (1999). *q*-gram based database searching using a suffix array (QUASAR). In *3rd RECOMB*, pages 77–83.

[28] Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.

[29] Carlson, J. M., Chakravarty, A., and Gross, R. H. (2006a). Beam: a beam search algorithm for the identification of cis-regulatory elements in groups of genes. *Journal of Computational Biology*, 13(3):686–701.

[30] Carlson, J. M., Chakravarty, A., Khetani, R. S., and Gross, R. H. (2006b). Bounded search for de novo identification of degenerate cis-regulatory elements. *BMC bioinformatics*, 7(1):254.

[31] Carvalho, A. M., Freitas, A. T., Oliveira, A. L., and Sagot, M. (2006a). An efficient algorithm for the identification of structured motifs in DNA promoter sequences. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(2):126–140.

[32] Carvalho, A. M., Freitas, A. T., Oliveira, A. L., and Sagot, M.-F. (2004). Efficient extraction of structured motifs using box-links. In *International Symposium on String Processing and Information Retrieval*, pages 267–268. Springer.

[33] Carvalho, A. M., Freitas, A. T., Oliveira, A. L., and Sagot, M.-F. (2005). A highly scalable algorithm for the extraction of cis-regulatory regions. In *Proceedings Of The 3rd Asia-Pacific Bioinformatics Conference*, pages 273–282. World Scientific.

[34] Carvalho, A. M., Freitas, A. T., Oliveira, A. L., and Sagot, M.-F. (2006b). An efficient algorithm for the identification of structured motifs in dna promoter sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 3(2):126–140.

[35] Chamary, J., Parmley, J. L., and Hurst, L. D. (2006). Hearing silence: non-neutral evolution at synonymous sites in mammals. *Nature Reviews Genetics*, 7(2):98.

[36] Cisłak, A., Grabowski, S., Holub, J., and Birol, I. (2018). Sopang: online text searching over a pan-genome. *Bioinformatics*.

[37] Cohen, S., Houben, A., and Segal, D. (2008). Extrachromosomal circular DNA derived from tandemly repeated genomic sequences in plants. *Plant J*, 53(6):1027–1034.

[38] Consortium, T. C. P.-G. (2016). Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, page 1–18.

[39] Consortium, U. et al. (2014). Uniprot: a hub for protein information. *Nucleic acids research*, page gku989.

[40] Cooke, M. S., Evans, M. D., Dizdaroglu, M., and Lunec, J. (2003). Oxidative dna damage: mechanisms, mutation, and disease. *The FASEB Journal*, 17(10):1195–1214.

[41] Craik, D. J. and Allewell, N. M. (2012). Thematic minireview series on circular proteins. *J Biol Chem*, 287(32):26999–27000.

[42] Crick, F. H. (1968). The origin of the genetic code. *Journal of molecular biology*, 38(3):367–379.

[43] Crochemore, M., Hancart, C., and Lecroq, T. (2007a). *Algorithms on strings*. Cambridge University Press.

[44] Crochemore, M., Hancart, C., and Lecroq, T. (2007b). *Algorithms on Strings*. Cambridge University Press, New York, NY, USA.

[45] Crochemore, M., Iliopoulos, C. S., Mohamed, M., and Sagot, M.-F. (2006). Longest repeats with a block of k don't cares. *Theoretical Computer Science*, 362(1-3):248–254.

[46] Dahiya, A. and Garg, D. (2014). Maximal pattern matching with flexible wildcard gaps and one-off constraint. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 1107–1112. IEEE.

[47] De Bruijn, N. G. (1946). A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764.

[48] Del Castillo, C. S., Hikima, J.-i., Jang, H.-B., Nho, S.-W., Jung, T.-S., Wongtavatchai, J., Kondo, H., Hirono, I., Takeyama, H., and Aoki, T. (2013). Comparative sequence analysis of a multidrug-resistant plasmid from Aeromonas hydrophila. *Antimicrob Agents Chemother*, 57(1):120–129.

[49] D'haeseleer, P. (2006). How does dna sequence motif discovery work? *Nature biotechnology*, 24(8):959.

[50] Do, H. H., Jansson, J., Sadakane, K., and Sung, W.-K. (2014). Fast relative lempel–ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30.

[51] Ehlers, T., Manea, F., Mercaş, R., and Nowotka, D. (2014). k-abelian pattern matching. In *18th DLT*, volume 8633 of *LNCS*, pages 178–190.

[52] Erill, I. and O'Neill, M. C. (2009). A reexamination of information theory-based methods for dna-binding site identification. *BMC bioinformatics*, 10(1):57.

[53] Eskin, E. and Pevzner, P. A. (2002). Finding composite regulatory patterns in dna sequences. *Bioinformatics*, 18(suppl_1):S354–S363.

[54] Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS*, pages 137–143.

[55] Feng, D.-F. and Doolittle, R. F. (1987). Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of molecular evolution*, 25(4):351–360.

[56] Fernandes, F., Pereira, L., and Freitas, A. T. (2009). CSA: An efficient algorithm to improve circular DNA multiple alignment. *BMC Bioinform*, 10(1):1–13.

[57] Ferrada, H., Gagie, T., Hirvola, T., and Puglisi, S. J. (2014). Hybrid indexes for repetitive datasets. *Phil. Trans. R. Soc. A*, 372(2016):20130137.

[58] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE.

[59] Fischer, J. (2011). Inducing the LCP-Array. In *12th WADS*, volume 6844 of *LNCS*, pages 374–385.

[60] Fletcher, W. and Yang, Z. (2009). INDELible: A flexible simulator of biological sequence evolution. *Mol Biol Evol*, 26(8):1879–1888.

[61] Frith, M. C., Saunders, N. F., Kobe, B., and Bailey, T. L. (2008). Discovering sequence motifs with arbitrary insertions and deletions. *PLoS computational biology*, 4(5):e1000071.

[62] Fuller, R. S., Funnell, B. E., and Kornberg, A. (1984). The dnaa protein complex with the e. coli chromosomal replication origin (oric) and other dna sites. *Cell*, 38(3):889–900.

[63] Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337.

[64] Goios, A., Pereira, L., Bogue, M., Macaulay, V., and Amorim, A. (2007). mtDNA phylogeny and evolution of laboratory mouse strains. *Genome Res*, 17(3):293–298.

[65] Gotoh, O. (1982). An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708.

[66] Gray, F. (1953). Pulse code communication. *US Patent 2632058*.

[67] Grossi, R., Iliopoulos, C. S., Liu, C., Pisanti, N., Pissis, S. P., Retha, A., Rosone, G., Vayani, F., and Versari, L. (2017a). On-line pattern matching on similar texts. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 78. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[68] Grossi, R., Iliopoulos, C. S., Mercas, R., Pisanti, N., Pissis, S. P., Retha, A., and Vayani, F. (2015). Circular sequence comparison with q-grams. In Pop, M. and Touzet, H., editors, *Algorithms in Bioinformatics - 15th International Workshop, WABI 2015, Atlanta, GA, USA, September 10-12, 2015, Proceedings*, volume 9289 of *Lecture Notes in Computer Science*, pages 203–216. Springer.

[69] Grossi, R., Iliopoulos, C. S., Mercas, R., Pisanti, N., Pissis, S. P., Retha, A., and Vayani, F. (2016). Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11(1):12.

[70] Grossi, R., Menconi, G., Pisanti, N., Trani, R., and Vind, S. (2014). Output-sensitive pattern extraction in sequences. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[71] Grossi, R., Menconi, G., Pisanti, N., Trani, R., and Vind, S. (2017b). Motif trie: An efficient text index for pattern discovery with don't cares. *Theoretical Computer Science*.

[72] Grossi, R., Pietracaprina, A., Pisanti, N., Pucci, G., Upfal, E., and Vandin, F. (2011). Madmx: A strategy for maximal dense motif extraction. *Journal of Computational Biology*, 18(4):535–545.

[73] Gusfield, D. (1997a). *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press.

[74] Gusfield, D. (1997b). *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press.

[75] Helinski, D. R. and Clewell, D. B. (1971). Circular DNA. *Annu Rev Biochem*, 40(1):899–942.

[76] Henikoff, S., Henikoff, J. G., Alford, W. J., and Pietrokovski, S. (1995). Automated construction and graphical presentation of protein blocks from unaligned sequences. *Gene*, 163(2):GC17–GC26.

[77] Hertz, G. Z. and Stormo, G. D. (1999). Identifying dna and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics (Oxford, England)*, 15(7):563–577.

[78] Holley, G., Wittler, R., and Stoye, J. (2016). Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11:3.

[79] Holub, J., Smyth, W. F., and Wang, S. (2008). Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50.

[80] Hu, J., Li, B., and Kihara, D. (2005). Limitations and potentials of current motif discovery algorithms. *Nucleic acids research*, 33(15):4899–4913.

[81] Huang, L., Popic, V., and Batzoglou, S. (2013). Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370.

[82] Huerta-Cepas, J., Serra, F., and Bork, P. (2016). Ete 3: reconstruction, analysis, and visualization of phylogenomic data. *Molecular biology and evolution*, 33(6):1635–1638.

[83] Iliopoulos, C. S., Kundu, R., and Pissis, S. P. (2017). Efficient pattern matching in elastic-degenerate texts. In *11th International Conference on Language and Automata Theory and Applications (LATA)*, volume 10168 of *Lecture Notes in Computer Science*. Springer International Publishing.

[84] Iliopoulos, C. S., McHugh, J., Peterlongo, P., Pisanti, N., Rytter, W., and Sagot, M.-F. (2005). A first approach to finding common motifs with gaps. *International Journal of Foundations of Computer Science*, 16(06):1145–1154.

[85] Iliopoulos, C. S., Mohamed, M., Pissis, S. P., and Vayani, F. (2018). Maximal motif discovery in a sliding window. In *International Symposium on String Processing and Information Retrieval*, pages 191–205. Springer.

[86] IUPAC-IUB Commission on Biochemical Nomenclature (1971). Abbreviations and symbols for nucleic acids, polynucleotides and their constituents. *Archives of Biochemistry and Biophysics*.

[87] Jensen, K. L., Styczynski, M. P., Rigoutsos, I., and Stephanopoulos, G. N. (2005). A generic motif discovery algorithm for sequential data. *Bioinformatics*, 22(1):21–28.

[88] Jonassen, I., Collins, J. F., and Higgins, D. G. (1995). Finding flexible patterns in unaligned protein sequences. *Protein science*, 4(8):1587–1595.

[89] Kawai, Y., Saito, T., Kitazawa, H., and Itoh, T. (1998). Gassericin a; an uncommon cyclic bacteriocin produced by lactobacillus gasseri la39 linked at n-and c-terminal ends. *Bioscience, biotechnology, and biochemistry*, 62(12):2438–2440.

[90] Keich, U. and Pevzner, P. A. (2002). Finding motifs in the twilight zone. In *Proceedings of the sixth annual international conference on Computational biology*, pages 195–204. ACM.

[91] Kemperman, R., Kuipers, A., Karsens, H., Nauta, A., Kuipers, O., and Kok, J. (2003). Identification and characterization of two novel clostridial bacteriocins, circularin a and closticin 574. *Applied and environmental microbiology*, 69(3):1589–1597.

[92] Kersey, P. J., Allen, J. E., Armean, I., Boddu, S., Bolt, B. J., Carvalho-Silva, D., Christensen, M., Davis, P., Falin, L. J., Grabmueller, C., Humphrey, J. C., Kerhornou, A., Khobova, J., Aranganathan, N. K., Langridge, N., Lowy, E., McDowall, M. D., Maheswari, U., Nuhn, M., Ong, C. K., Overduin, B., Paulini, M., Pedro, H., Perry, E., Spudich, G., Tapanari, E., Walts, B., Williams, G., Tello-Ruiz, M. K., Stein, J. C., Wei, S., Ware, D., Bolser, D. M., Howe, K. L., Kulesha, E., Lawson, D., Maslen, G., and Staines, D. M. (2016). Ensembl genomes 2016: more genomes, more complexity. *Nucleic Acids Research*, 44(Database-Issue):574–580.

[93] Kiesel, A., Roth, C., Ge, W., Wess, M., Meier, M., and Söding, J. (2018). The bamm web server for de-novo motif discovery and regulatory sequence analysis. *Nucleic acids research*.

[94] Knuth, D. E., Jr., J. H. M., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350.

[95] Kreft, S. and Navarro, G. (2013). On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133.

[96] Kuruppu, S., Puglisi, S. J., and Zobel, J. (2010). Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *International Symposium on String Processing and Information Retrieval*, pages 201–206. Springer.

[97] Kuttler, F. and Mai, S. (2007). Formation of non-random extrachromosomal elements during development, differentiation and oncogenesis. In *Seminars in cancer biology*, volume 17, pages 56–64. Elsevier.

[98] Lawrence, C. E., Altschul, S. F., Boguski, M. S., Liu, J. S., Neuwald, A. F., and Wootton, J. C. (1993). Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *science*, 262(5131):208–214.

[99] Lee, T., Na, J. C., Park, H., Park, K., and Sim, J. S. (2013). Finding consensus and optimal alignment of circular strings. *Theor Comput Sci*, 468:92–101.

[100] Leonard, A. C. and Méchali, M. (2013). Dna replication origins. *Cold Spring Harbor perspectives in biology*, 5(10):a010116.

[101] Leung, H. C. and Chin, F. Y. (2006). An efficient motif discovery algorithm with unknown motif length and number of binding sites. *International journal of data mining and bioinformatics*, 1(2):201–215.

[102] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[103] Li, N. and Tompa, M. (2006). Analysis of computational approaches for motif discovery. *Algorithms for molecular biology*, 1(1):8.

[104] Luo, H., Zhang, C.-T., and Gao, F. (2014). Ori-finder 2, an integrated tool to predict replication origins in the archaeal genomes. *Frontiers in microbiology*, 5:482.

[105] Maciuca, S., del Ojo Elias, C., McVean, G., and Iqbal, Z. (2016). A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *16th International Conference on Algorithms in Bioinformatics (WABI)*, pages 222–233.

[106] Maes, M. (1990). On a cyclic string-to-string correction problem. *IPL*, 35(2):73–78.

[107] Mäkinen, V., Navarro, G., Sirén, J., and Välimäki, N. (2010). Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308.

[108] Manber, U. and Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM J Comput*, 22(5):935–948.

[109] Marzal, A. and Barrachina, S. (2000). Speeding up the computation of the edit distance for cyclic strings. In *15th ICPR*, volume 2, pages 891–894.

[110] McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272.

[111] Meijer, M., Beck, E., Hansen, F. G., Bergmans, H., Messer, W., Von Meyenburg, K., and Schaller, H. (1979). Nucleotide sequence of the origin of replication of the escherichia coli k-12 chromosome. *Proceedings of the National Academy of Sciences*, 76(2):580–584.

[112] Mosig, A., Hofacker, I. L., and Stadler, P. F. (2006a). Comparative Analysis of Cyclic Sequences: Viroids and other Small Circular RNAs. In *GCB*, volume 83 of *LNI*, pages 93–102. GI.

[113] Mosig, A., Hofacker, I. L., and Stadler, P. F. (2006b). Comparative Analysis of Cyclic Sequences: Viroids and other Small Circular RNAs. In *GCB*, volume 83 of *LNI*, pages 93–102. GI.

[114] Na, J. C., Kim, H., Park, H., Lecroq, T., Léonard, M., Mouchard, L., and Park, K. (2016). Fm-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170.

[115] Nabholz, B., Glémin, S., and Galtier, N. (2007). Strong variations of mitochondrial mutation rate across mammals—the longevity hypothesis. *Molecular biology and evolution*, 25(1):120–130.

[116] Nabiyouni, M., Prakash, A., and Fedorov, A. (2013). Vertebrate codon bias indicates a highly gc-rich ancestral genome. *Gene*, 519(1):113–119.

[117] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453.

[118] Nguyen, N., Hickey, G., Zerbino, D. R., Raney, B. J., Earl, D., Armstrong, J., Kent, W. J., Haussler, D., and Paten, B. (2015). Building a pan-genome reference for a population. *Journal of Computational Biology*, 22(5):387–401.

[119] Parida, L., Pizzi, C., and Rombo, S. E. (2014). Irredundant tandem motifs. *Theoretical Computer Science*, 525:89–102.

[120] Parida, L., Rigoutsos, I., Floratos, A., Platt, D. E., and Gao, Y. (2000). Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm. In *SODA*, pages 297–308.

[121] Parida, L., Rigoutsos, I., and Platt, D. (2001). An output-sensitive flexible pattern discovery algorithm. In *Annual Symposium on Combinatorial Pattern Matching*, pages 131–142. Springer.

[122] Pavesi, G., Mereghetti, P., Mauri, G., and Pesole, G. (2004). Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. *Nucleic Acids Research*, 32(Web-Server-Issue):199–203.

[123] Peterlongo, P., Pisanti, N., Boyer, F., do Lago, A. P., and Sagot, M.-F. (2008). Lossless filter for multiple repetitions with Hamming distance. *JDA*, 6(3):497–509.

[124] Peterlongo, P., Sacomoto, G. T., do Lago, A. P., Pisanti, N., and Sagot, M.-F. (2009). Lossless filter for multiple repeats with bounded edit distance. *Algorithm Mol Biol*, 4(3).

[125] Pevzner, P. A., Sze, S.-H., et al. (2000). Combinatorial approaches to finding subtle signals in dna sequences. In *ISMB*, volume 8, pages 269–278.

[126] Pisanti, N., Carvalho, A. M., Marsan, L., and Sagot, M. (2006a). RISOTTO: fast extraction of motifs with mismatches. In Correa, J. R., Hevia, A., and Kiwi, M. A., editors, *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 757–768. Springer.

[127] Pisanti, N., Carvalho, A. M., Marsan, L., and Sagot, M.-F. (2006b). Risotto: Fast extraction of motifs with mismatches. In *Latin American Symposium on Theoretical Informatics*, pages 757–768. Springer.

[128] Pisanti, N., Crochemore, M., Grossi, R., and Sagot, M.-F. (2003). A basis of tiling motifs for generating repeated patterns and its complexity for higher quorum. In *International Symposium on Mathematical Foundations of Computer Science*, pages 622–631. Springer.

[129] Pisanti, N., Crochemore, M., Grossi, R., and Sagot, M.-F. (2005). Bases of motifs for generating repeated patterns with wild cards. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(1):40–50.

[130] Pisanti, N., Giraud, M., and Peterlongo, P. (2010). Filters and seeds approaches for fast homology searches in large datasets. In Elloumi, M. and Zomaya, A. Y., editors, *Algorithms in computational molecular biology*, chapter 15, pages 299–320. John Wiley & sons.

[131] Pisanti, N., Soldano, H., Carpentier, M., and Pothier, J. (2009). A relational extension of the notion of motifs: Application to the common 3D protein substructures searching problem. *Journal of Computational Biology*, 16(12):1635–1660.

[132] Pissis, S. P. (2014). MoTeX-II: structured MoTif eXtraction from large-scale datasets. *BMC Bioinformatics*, 15:235.

[133] Pissis, S. P. and Retha, A. (2018). Dictionary matching in elastic-degenerate texts with applications in searching vcf files on-line. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 103. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[134] Pissis, S. P., Stamatakis, A., and Pavlidis, P. (2013). MoTeX: A word-based HPC tool for motif extraction. In Gao, J., editor, *ACM Conference on Bioinformatics, Computational Biology and Biomedical Informatics. ACM-BCB 2013, Washington, DC, USA, September 22-25, 2013*, page 13. ACM.

[135] Ponting, C. P. and Russell, R. B. (1995). Swaposins: circular permutations within genes encoding saposin homologues. *Trends Biochem Sci*, 20(5):179–180.

[136] Price, A., Ramabhadran, S., and Pevzner, P. A. (2003). Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19(suppl_2):ii149–ii155.

[137] Pruitt, K. D., Tatusova, T., and Maglott, D. R. (2007). Ncbi reference sequences (refseq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic acids research*, 35(suppl 1):D61–D65.

[138] Rasmussen, K., Stoye, J., and Myers, E. (2006). Efficient q-gram Filters for finding all epsilon-matches over a given length. *J Comput Biol*, 13(2):296–308.

[139] Rice, P., Longden, I., and Bleasby, A. (2000). EMBOSS: The European Molecular Biology Open Software Suite. *Trends Genet*, 16(6):276–277.

[140] Rigoutsos, I. and Floratos, A. (1998). Motif discovery without alignment or enumeration. In *Proceedings of the second annual international conference on Computational molecular biology*, pages 221–227. ACM.

[141] Rigoutsos, I., Floratos, A., Ouzounis, C., Gao, Y., and Parida, L. (1999). Dictionary building via unsupervised hierarchical motif discovery in the sequence space of natural proteins. *Proteins: Structure, Function, and Bioinformatics*, 37(2):264–277.

[142] Roberts, R. J., Belfort, M., Bestor, T., Bhagwat, A. S., Bickle, T. A., Bitinaite, J., Blumenthal, R. M., Degtyarev, S. K., Dryden, D. T. F., Dybvig, K., Firman, K., Gromova, E. S., Gumport, R. I., Halford, S. E., Hattman, S., Heitman, J., Hornby, D. P., Janulaitis, A., Jeltsch, A., Josephsen, J., Kiss, A., Klaenhammer, T. R., Kobayashi, I., Kong, H., Krüger, D. H., Lacks, S., Marinus, M. G., Miyahara, M., Morgan, R. D., Murray, N. E., Nagaraja, V., Piekarowicz, A., Pingoud, A., Raleigh, E., Rao, D. N., Reich, N., Repin, V. E., Selker, E. U., Shaw, P., Stein, D. C., Stoddard, B. L., Szybalski, W., Trautner, T. A., Van Etten, J. L., Vitor, J. M. B., Wilson, G. G., and Xu, S. (2003). A nomenclature for restriction enzymes, dna methyltransferases, homing endonucleases and their genes. *Nucleic Acids Research*, 31(7):1805–1812.

[143] Robinson, D. and Foulds, L. R. (1981). Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147.

[144] Rojas, A. and Romeu, A. (1996). A sequence analysis of the β-glucosidase sub-family b. *FEBS letters*, 378(1):93–97.

[145] Sagot, M., Viari, A., Pothier, J., and Soldano, H. (1995). Finding flexible patterns in a text: an application to three-dimensional molecular matching. *Computer Applications in the Biosciences*, 11(1):59–70.

[146] Sagot, M.-F. (1998). Spelling approximate repeated or common motifs using a suffix tree. In *Latin American Symposium on Theoretical Informatics*, pages 374–390. Springer.

[147] Sandelin, A., Alkema, W., Engström, P., Wasserman, W. W., and Lenhard, B. (2004). Jaspar: an open-access database for eukaryotic transcription factor binding profiles. *Nucleic acids research*, 32(suppl_1):D91–D94.

[148] Sandve, G. K. and Drabløs, F. (2006). A survey of motif discovery methods in an integrated framework. *Biology direct*, 1(1):11.

[149] Sanger, F., Nicklen, S., and Coulson, A. R. (1977). Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467.

[150] Sankoff, D., Leduc, G., Antoine, N., Paquin, B., Lang, B. F., and Cedergren, R. (1992). Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. *Proceedings of the National Academy of Sciences*, 89(14):6575–6579.

[151] Scherer, P. E., Okamoto, T., Chun, M., Nishimoto, I., Lodish, H. F., and Lisanti, M. P. (1996). Identification, sequence, and expression of caveolin-2 defines a caveolin gene family. *Proceedings of the National Academy of Sciences*, 93(1):131–135.

[152] Schieber, B. and Vishkin, U. (1988). On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262.

[153] Schwartz, D. and Gygi, S. P. (2005). An iterative statistical approach to the identification of protein phosphorylation motifs from large-scale data sets. *Nature biotechnology*, 23(11):1391.

[154] Senecoff, J. F., Rossmeissl, P. J., and Cox, M. M. (1988). Dna recognition by the flp recombinase of the yeast 2 μ plasmid. *Journal of Molecular Biology*, 201(2):405–421.

[155] Senft, M. (2005). Suffix tree for a sliding window: An overview. In *WDS*, volume 5, pages 41–46.

[156] Sheikhizadeh, S., Schranz, M. E., Akdel, M., de Ridder, D., and Smit, S. (2016). Pantools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):487–493.

[157] Sigrist, C. J. A., De Castro, E., Cerutti, L., Cuche, B. A., Hulo, N., Bridge, A., Bougueleret, L., and Xenarios, I. (2013). New and continuing developments at prosite. *Nucleic Acids Research*, 41(D1):344–347.

[158] Sinha, S. and Tompa, M. (2003). YMF: a program for discovery of novel transcription factor binding sites by statistical overrepresentation. *Nucleic Acids Research*, 31(13):3586–3588.

[159] Sirén, J. (2017). Indexing variation graphs. In Fekete, S. P. and Ramachandran, V., editors, *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.*, pages 13–27. SIAM.

[160] Sirén, J., Välimäki, N., and Mäkinen, V. (2014). Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(2):375–388.

[161] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

[162] Stankevicius, K., Lubys, A., Timinskas, A., Vaitkevicius, D., and Janulaitis, A. (1998). Cloning and analysis of the four genes coding for bpu 10i restriction—modification enzymes. *Nucleic acids research*, 26(4):1084–1091.

[163] Sukumaran, J. and Holder, M. T. (2010). Dendropy: a python library for phylogenetic computing. *Bioinformatics*, 26(12):1569–1571.

[164] Taanman, J.-W. (1999). The mitochondrial genome: structure, transcription, translation and replication. *Biochimica et Biophysica Acta (BBA)-Bioenergetics*, 1410(2):103–123.

[165] Tang, Z., Scherer, P. E., Okamoto, T., Song, K., Chu, C., Kohtz, D. S., Nishimoto, I., Lodish, H. F., and Lisanti, M. P. (1996). Molecular cloning of caveolin-3, a novel member of the caveolin gene family expressed predominantly in muscle. *Journal of Biological Chemistry*, 271(4):2255–2261.

[166] Tettelin, H., Masignani, V., Cieslewicz, M. J., Donati, C., Medini, D., Ward, N. L., Angiuoli, S. V., Crabtree, J., Jones, A. L., Durkin, A. S., et al. (2005). Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: implications for the microbial "pan-genome". *Proceedings of the National Academy of Sciences*, 102(39):13950–13955.

[167] The 1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, 526(7571):68–74.

[168] Tompa, M., Li, N., Bailey, T. L., Church, G. M., De Moor, B., Eskin, E., Favorov, A. V., Frith, M. C., Fu, Y., Kent, W. J., et al. (2005). Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 23(1):137.

[169] Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theor Comput Sci*, 92(1):191–211.

[170] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

[171] Vanet, A., Marsan, L., Labigne, A., and Sagot, M.-F. (2000). Inferring regulatory elements from a whole genome. an analysis of helicobacter pyloriσ80 family of promoter signals1. *Journal of molecular biology*, 297(2):335–353.

[172] Vanet, A., Marsan, L., and Sagot, M.-F. (1999). Promoter sequences and algorithmical methods for identifying them. *Research in Microbiology*, 150(9-10):779–799.

[173] Wang, C. K., Kaas, Q., Chiche, L., and Craik, D. J. (2008). Cybase: a database of cyclic protein sequences and structures, with applications in protein discovery and engineering. *Nucleic acids research*, 36(suppl 1):D206–D210.

[174] Wang, Z. and Wu, M. (2014). Phylogenomic reconstruction indicates mitochondrial ancestor was an energy parasite. *PLoS ONE*, 10(9):e110685.

[175] Waterman, M. S. (1984). General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46(4):473–500.

[176] Watson, J. D., Crick, F. H., et al. (1953). Molecular structure of nucleic acids. *Nature*, 171(4356):737–738.

[177] Watterson, G., Ewens, W. J., Hall, T. E., and Morgan, A. (1982). The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7.

[178] Weiner, J. and Bornberg-Bauer, E. (2006). Evolution of circular permutations in multidomain proteins. *Mol Biol Evol*, 23(4):734–743.

[179] Weiner, P. (1973). Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT)*, SWAT '73, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[180] Wheeler, T. J. (2009). Large-scale neighbor-joining with ninja. In *Algorithms in Bioinformatics*, pages 375–389. Springer.

[181] Wingender, E., Dietze, P., Karas, H., and Knüppel, R. (1996). Transfac: a database on transcription factors and their dna binding sites. *Nucleic acids research*, 24(1):238–241.

[182] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343.

Appendix A

Pseudocode

A.1 Right-Hand Side

```
1  Function 3 rightHandSide(i)
2      𝒜 ← readLetter(i);
3      switch 𝒜 = ⟨γ, μ, λ⟩ do
4          case 𝒜 = ⟨U, 1, i⟩ do
5              score ← createActivePoint(score);
6          case 𝒜 = ⟨U, 0, i⟩ do
7              score ← deleteActivePoint(score);
8          case 𝒜 = ⟨U_𝒜, 0, i⟩ do
9              U_𝒜 ← copySeedRHS(𝒜, U_𝒜);
10             E′ ← checkEndArray(j);
11             U ← s(U_𝒜);
12             𝒜 ← ⟨U, 0, i⟩;
13             j ++;
14             score, E' ← updateNodes(score, U, E');
15             if j > ℓ then
16                 score ← leftHandSide(score, j);
17             while j is being incremented do
18                 𝒜 ← ⟨s(U), 0, i⟩;
19                 j ++;
20                 E' ← updateEndArray(E');
21                 score, E' ← updateNodes(score, s(U), E');
22                 if j > ℓ then
23                     score ← leftHandSide(score, ST, j);
24     return;
```

```
1  Function 4 createActivePoint(score)
2      isSeed(𝒜) ← TRUE;
3      |occ(𝒜)| ← |occ(explicitChild(𝒜))| + 1;
4      if |occ(𝒜)| ≥ k then
5          initialise ℬ(𝒜) and pivot(𝒜) using explicitChild(𝒜);
6          E' ← getEndPositions();
7          if U = r then
8              isMotif(𝒜) ← FALSE;
9              score ← createEdges(score, E', 𝒜);
10         else
11             isMotif(𝒜) ← isMotif(U);
12             score ← transferEdges(score, U, E');
13         score ← updateSeedScores(𝒜, score);
14     return score;
```

```
1  Function 5 createEdges(score, E', U)
2      foreach ⟨V, j'⟩ ∈ E' do
3          d' ← j − j' − 1;
4          if popCount(ℬ(V −d'→ U) < k) then
5              delete ⟨V, j'⟩ from E';
6      cluster E' w.r.t. |occ(V ⋄^d' U)|;
7      foreach cluster c ∈ E' do
8          sort c w.r.t D(V);
9          V ← deepest node in c;
10         if ∄ V' → U : descendant(V) = V' & ℬ(V' → U) ≡ ℬ(V −d'→ U) then
11             if ∄ V −d'→ U then
12                 add V −d'→ U to motif graph;
13                 if indegree(V) = 0 & |occ(U)| ≠ |occ(V ⋄^d' U)| then
14                     score ++;
15                 else
16                     score ← checkPaths(score, V −d'→ U);
17             else
18                 score ← updatePaths(score, V −d'→ U);
20     return score;
```

```
1  Function 6 checkPaths(score, V −d'→ U)
2      foreach prefix of a path p ending with V −d'→ U do
3          if ℬ(prefix(p)) ≢ ℬ(p)) then
4              score ++;
5      return score;
```

```
1 Function 7 updatePaths(score, V -d'-> U)
2     foreach prefix of a path p ending with V -d'-> U do
3         if (B(prefix(p)) ≡ B(p)) & (∄ a path p' : B(prefix(p)) ≡ B(suffix(p'))) then
4             score - -;
5     return score;
```

```
1  Function 8 updateSeedScores(score, U)
2      if isMotif(U) = FALSE then
3          if |occ(U)| = k & indegree(U) = 0 then
               ▷ |occ(U)| just hit threshold k but no edges were added
4              score ++;
5              isMotif(U) ← TRUE;
6          else if
             (|occ(U)| > k) & ((indegree(U) > 1) or (indegree(U) = 1 & B(V -d'-> U) ≢ B(U)))
            then
               ▷ let V -d'-> U be the only incoming edge to U
7              score ++;
8              isMotif(U) ← TRUE;
9      else
10         foreach edge V -d'-> U recently added to motif graph do
11             if B(U) ≡ B(V -d'-> U) then
12                 isMotif(U) ← FALSE;
13                 score - -;
14                 breakForLoop;
16     foreach edge V -d'-> U recently added to motif graph do
17         if isMotif(V) = TRUE & (B(V) ≡ B(V -d'-> U)) then
18             isMotif(V) ← FALSE;
19             score = score − 1;
20             breakForLoop;
21     foreach edge V -d'-> U recently added to motif graph do
22         if isMotif(V) = FALSE then
23             if (|occ(V)| > k) & ((outdegree(V) > 1) or (outdegree(V) = 1 & B(V -d'-> U) ≢
                B(V))) then
                   ▷ let V -d'-> U be the only outgoing edge from V
24                 score ++;
25                 isMotif(V) ← TRUE;
26     return score;
```

```
1  Function 9 deleteActivePoint()
2      |occ(U)| + +;
3      update isSeed(U);
4      if isSeed(U) = TRUE & |occ(U)| ≥ k then
5          B(U) ← B(A);
6          pivot(U) ← pivot(A);
7          add pointer to U to the list of pointers at E[j + D(U) − 1];
8          score ← mergeEdges(score, A, U);
9          score ← updateSeedScores(U, score);
10     if isMotif(A) = TRUE then
11         score - -;
12     delete pivot(A), B(A), |occ(A)| and all motif edges;
```

```
1  Function 10 mergeEdges(score, A, U)
2      foreach edge V --d'--> A do
3          if ∃ V --d'--> U then
4              foreach path ending with V --d'--> A do
5                  score - -;
6          else
7              create V --d'--> U;
8      return score, isMotif(U);
```

Function ***11*** *transferEdges*$(score, U, E')$

foreach $\langle V, j' \rangle \in E'$ **do**

$d' \leftarrow j - j' - 1$;

if $(\mathscr{B}(V) \equiv \mathscr{B}(\mathscr{A}))$ **or** $kPopCount(k, \mathscr{B}(V \xrightarrow{d'} \mathscr{A})) = \text{TRUE}$ **then**

add $V \xrightarrow{d'} \mathscr{A}$;

foreach *path ending with* $V \xrightarrow{d'} \mathscr{A}$ **do**

$score$ + +;

if $\mathscr{B}(V \xrightarrow{d'} \mathscr{A}) \equiv \mathscr{B}(V \xrightarrow{d'} U)$ **then**

delete $V \xrightarrow{d'} U$;

foreach *path ending with* $V \xrightarrow{d'} U$ **do**

$score$ - -;

return *score*;

Function ***12*** *copySeedRHS*(V, U)

$\mathsf{isSeed}(U) \leftarrow \text{TRUE}$;

update $\mathscr{B}(U)$ and pivot(U) using V;

$\mathsf{isMotif}(U) \leftarrow \mathsf{isMotif}(V)$;

move all outgoing edges from V to U;

return U;

```
1  Function 13 updateNodes(score, U, E')
2      foreach explicit non-root node U' in path from r to U, incl. U do
3          |occ(U')| + +;
4          update isSeed(U');
5          if isSeed(U') = true & |occ(U')| ≥ k then
6              update B(U') and pivot(U');
7              add pointer to U' to the list of pointers at E[j + D(U') − 1];
8              score ← createEdges(score, E', U');
9              foreach edge V --d'--> U' added to motif graph do
10                 delete corresponding cluster from E';
11             score ← updateSeedScore(score, U');
12     return score, E';
```

```
1  Function 14 updateEndArray(E')
2      foreach pair ⟨V, j'⟩ ∈ E' do
3          if j' = j − 1 − d then
4              delete ⟨V, j'⟩ from E';
5      foreach V in list at E[j − 1] do
6          j' ← j − 1;
7          add ⟨V, j'⟩ to E';
8      return E';
```

A.2 Left-Hand Side

```
1  Function 15 leftHandSide(score, ST, j)
2      V', 𝒜 ← deleteLetter(j, ST);
3      foreach internal node V in the path from V' to r, inclusive do
4          |occ(V)| - -;
5          update isSeed(V);
6          if s(V) ≠ r & isSeed(s(V)) = FALSE then
7              score, s(V) ← copySeedLHS(score, V, s(V));
8          else
9              if |occ(V)| = k − 1 or isSeed(V) = FALSE then
10                 foreach outgoing edge from V do
11                     score ← deleteRelevantEdge(score, edge);
12                 if isMotif(V) = TRUE then
13                     score - -;
14                 remove all pointers to V from array E;
15                 delete ℬ(V), pivot(V) and isMotif(V);
16             else
17                 foreach relevant outgoing motif edge from V do
                       ▷ let the destination of the motif edge be Z
18                     if s(V) ≠ r & isSeed(s(V)) = TRUE then
19                         score ← checkRelevantEdge(V, Z);
20                     else if (s(V) = r) & ((|occ(V)| = k & ℬ(V --d'--> Z) ≡
                           ℬ(V)) or (popCount(ℬ(V --d'--> Z)) = k)) then
21                         score ← deleteEdge(score, V --d'--> Z);
22                 shift ℬ(V) and pivot(V);
23                 score ← changeSeedScore(score, V);
24     if 𝒜 moved to a different branch then
25         rightHandSide(i, 𝒜);
26     print(score);
27     return
```

Function *16* *copySeedLHS*$(score, V, U)$

isSeed$(U) \leftarrow$ TRUE;
update pivot(U) using pivot(V);
update $\mathscr{B}(U)$ using $\mathscr{B}(V)$;
isMotif$(U) \leftarrow$ isMotif(V);
if *isMotif*$(U) =$ TRUE **then**
    $score$ ++;
copy all outgoing edges from V to U;
if $|occ(V)| \geq k$ **then**
    foreach *maximal path from* U **do**
        $score$ ++;
return $score, U$;

Function *17* *deleteRelevantEdge*$(score, edge)$

▷ let the destination node of the edge be Z, and the edge label be d'
if $s(V) \neq r$ & *isSeed*$(s(V)) =$ TRUE **then**
    if $\exists\, s(V) \xrightarrow{d'} Z$ **then**
        $score \leftarrow$ checkPathsFromSeedSuffix$(score, V, Z)$;
    else
        $score \leftarrow$ addEdge$(score, s(V) \xrightarrow{d'} Z)$;
$score \leftarrow$ deleteEdge$(score, V \xrightarrow{d'} Z)$;
return $score$;

Function *18* *checkPathsFromSeedSuffix*$(score, V, Z)$

foreach *maximal path p starting with* $V \xrightarrow{d'} Z$ **do**

if $\mathscr{B}(p) \not\equiv \mathscr{B}(p')$ **then**

$\triangleright\ p'$ is obtained by replacing $V \xrightarrow{d'} Z$ with $s(V) \xrightarrow{d'} Z$

score - -;

return *score*;

Function *19* *deleteEdge*$(score, edge)$

foreach *maximal path p starting with* $V \xrightarrow{d'} Z$ **do**

if $\mathscr{B}(p) \not\equiv \mathscr{B}(\mathit{suffix}(p))$ **then**

score - -;

delete *edge*;

if $\nexists\, p'$: $\mathscr{B}(\mathit{suffix}(p)) \equiv \mathscr{B}(\mathit{prefix}(p'))$ **then**

score ++;

return *score*;

Function *20* *addEdge*$(score, s(V) \xrightarrow{d'} U)$

add $s(V) \xrightarrow{d'} U$;

if $\mathit{isMotif}(U) = \text{TRUE}$ & $\mathscr{B}(U) \equiv \mathscr{B}(s(V) \xrightarrow{d'} U)$ **then**

$\mathsf{isMotif}(U) \leftarrow \text{FALSE}$;

score - -;

if $\mathit{isMotif}(s(V)) = \text{TRUE}$ & $\mathscr{B}(s(V)) \equiv \mathscr{B}(s(V) \xrightarrow{d'} U)$ **then**

$\mathsf{isMotif}(s(V)) \leftarrow \text{FALSE}$;

score - -;

return *score*;

```
1  Function 21 checkRelevantEdge(score, V, Z)
2      if ∃ s(V) --d'--> Z then
3          if popCount(V --d'--> Z) = k then
4              score ← checkPathsFromSeedSuffix(score, V, Z);
5              score ← deleteEdge(score, V --d'--> Z);
6      else
7          score ← addEdge(score, s(V) --d'--> Z);
8          if popCount(V --d'--> Z) = k then
9              score ← deleteEdge(score, V --d'--> Z);
10         else
11             foreach path p starting with V --d'--> Z do
                   ▷ let p' be the same path as p but with the first edge replaced by s(V) --d'--> Z
12                 if 𝓑(p) ≢ 𝓑(p') then
13                     score ++;
14     return score;
```

```
1  Function 22 changeSeedScore(score,V)
2      if outdegree(V) = 0 then
3          if isMotif(V) = FALSE then
4              isMotif(V) ← TRUE;
5              score ++;
6      else
7          flag ← FALSE;
8          foreach outgoing edge from V do
9              if B(edge) ≡ B(V) then
10                 flag ← TRUE;
11         if flag = TRUE & isMotif(V) = TRUE then
12             isMotif(V) ← FALSE;
13             score - -;
14         else if flag = FALSE & isMotif(V) = FALSE then
15             isMotif(V) ← TRUE;
16             score ++;
17     return score;
```

Appendix B

Codon Translation Table

The wheel below defines the amino acids resulting from the translation of all combinations of RNA codons. The RNA alphabet is similar to the DNA alphabet, where only `T` from the DNA alphabet is replaced by `U` in the RNA alphabet. The innermost nucleotide in the wheel is at the 5' end of the codon, and thus the outermost nucleotide is at the 3' end. The $\star$ symbol indicates the start codon and † indicates the stop codon; these special codons respectively initiate and terminate translation.